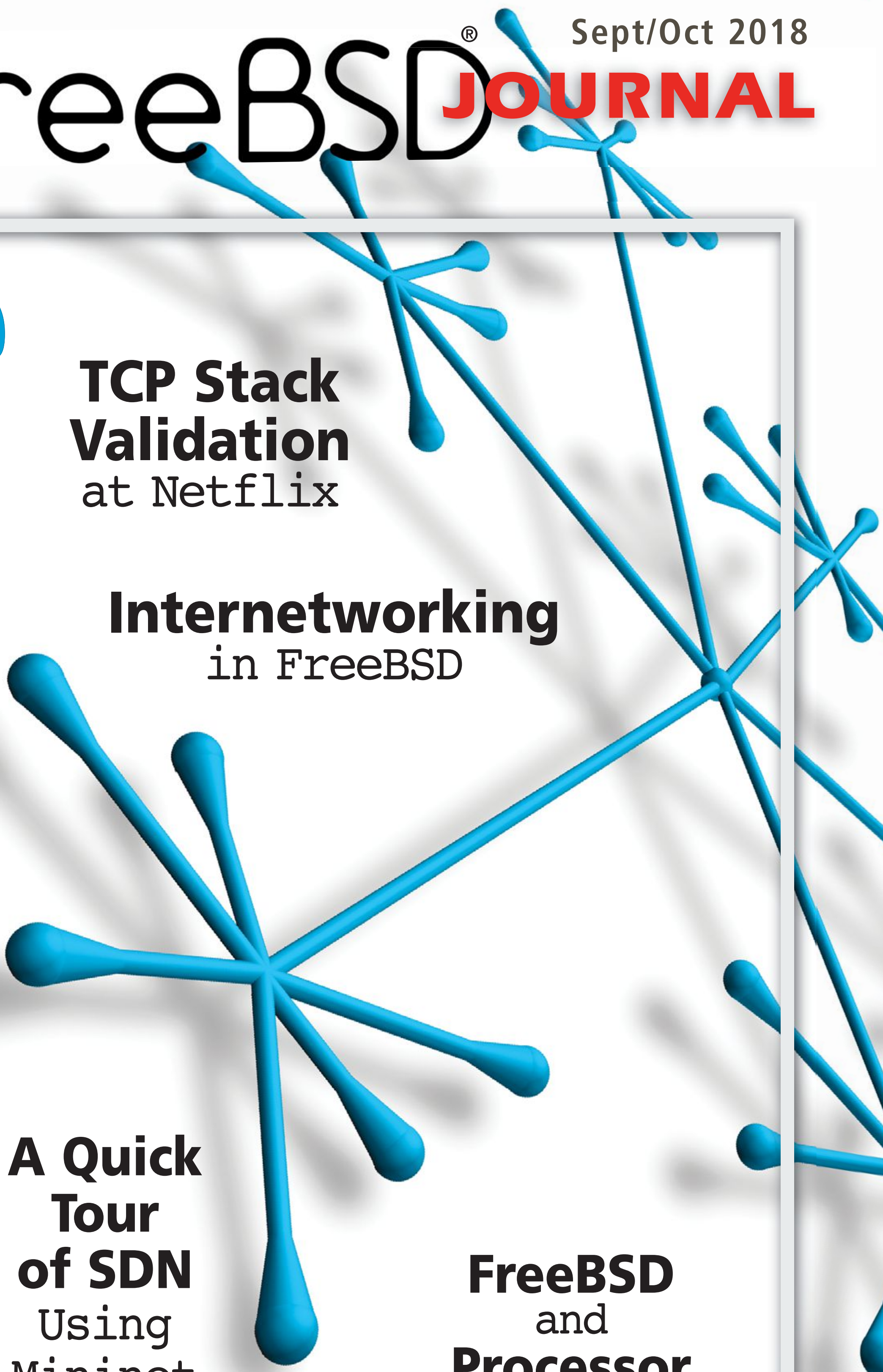
FreeBSD® JOURNAL

Sept/Oct 2018

networking



TCP Stack Validation at Netflix

Internetworking in FreeBSD

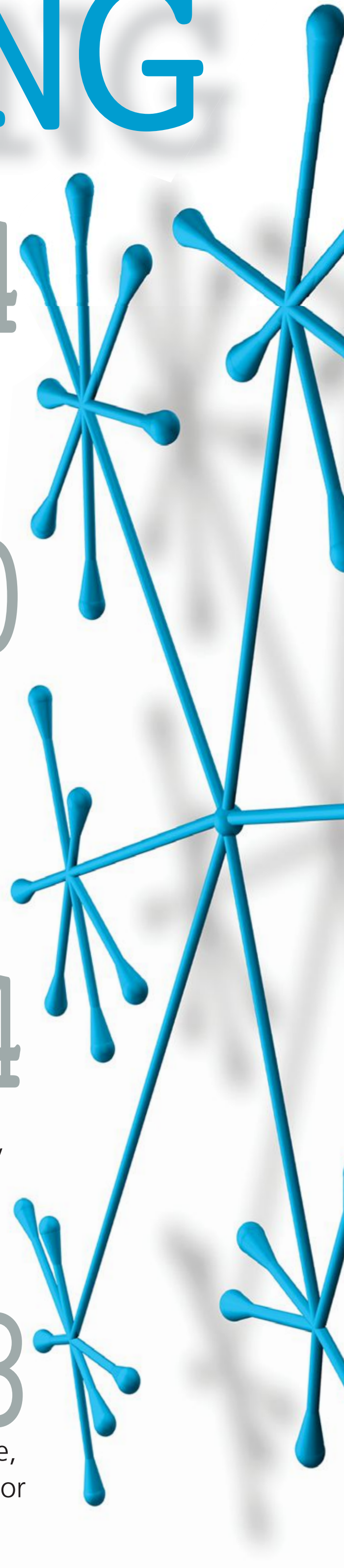
A Quick Tour of SDN Using Mininet

FreeBSD and **Processor TRENDS**

Table of Contents

Sept/Oct 2018

NETWORKING



Support FreeBSD®

Donate to the Foundation!

You already know that FreeBSD is an internationally recognized leader in providing a high-performance, secure, and stable operating system. It's because of you. Your donations have a direct impact on the Project.

Please consider making a gift to support FreeBSD for the coming year. It's only with your help that we can continue and increase our support to make FreeBSD the high-performance, secure, and reliable OS you know and love!

Your investment will help:

- Funding Projects to Advance FreeBSD
- Increasing Our FreeBSD Advocacy and Marketing Efforts
- Providing Additional Conference Resources and Travel Grants
- Continued Development of the FreeBSD Journal
- Protecting FreeBSD IP and Providing Legal Support to the Project
- Purchasing Hardware to Build and Improve FreeBSD Project Infrastructure

Making a donation is quick and easy.

freebsdfoundation.org/donate

FreeBSD™ FOUNDATION

S&W PUBLISHING LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@freebsdjournal.com

Editor-at-Large • James Maurer
jmaurer@freebsdjournal.com

Copy Editor • Annaliese Jakimides

Art Director • Dianne M. Kischitz
dianne@freebsdjournal.com

Advertising Sales • Walter Andrzejewski
walter@freebsdjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year (January/February, March/April, May/June, July/August, September/October, November/December).
Published by the FreeBSD Foundation,
5757 Central Ave., Suite 201, Boulder, CO 80301
ph: 720/207-5142 • fax: 720/222-2350
email: info@freebsdfoundation.org
Copyright © 2018 by FreeBSD Foundation. All rights reserved. This magazine may not be reproduced in whole or in part without written permission from the publisher.

LETTER from the Board

Welcome to *FreeBSD Journal*'s fall 2018 issue on Networking. The first of our networking pieces, an article by Jonathan Looney, covers a topic in which I've always had a strong interest, testing. As hard as it is to test software, testing network code, which effectively means testing a distributed system, is far more difficult. The folks at Netflix, which uses FreeBSD in their caching servers, have an interest in both performance and correctness of changes they're making in the FreeBSD TCP stack, and so the rigor of a good testing system is most welcome for them and for FreeBSD in general.

Ayaka Koshibe has contributed a piece on Software Defined Networking (SDN) and Mininet, a well-known emulator for SDN protocols. She has been presenting work around this topic at conferences over the last couple years, and I'm glad we were able to convince her to contribute an article on the topic to the *Journal*.

The third piece, "Internetworking with FreeBSD," is one your humble Editor in Chief contributed, so I'll not ruin the surprise by talking about it here.

To round out the section, we have an article by Matt Macy about how to scale the performance of FreeBSD on modern server hardware. While FreeBSD is deployed on many types of hardware and in many environments, performance on high-end servers, with many CPU cores and large memories, remains key to FreeBSD's overall success.

Please take a look at all our columns, particularly the new Letters column, where Michael Lucas answers your letters with delicacy and wit.

George Neville-Neil
President of the *FreeBSD Foundation* Board of Directors

SEE
TEXT
ONLY

INTERNETWORKING in FreeBSD

By George V. Neville-Neil

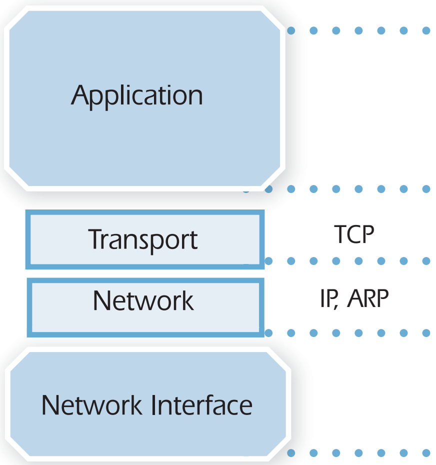
The FreeBSD network stack is the kernel software that, when taken together, allows programs on one computer to talk to programs on another. It is a highly reusable software artifact that has been employed not only within the operating system, but has been ported many times to other environments, including embedded operating systems completely unrelated to FreeBSD. For most developers, working both inside and outside the operating system's kernel, the network stack is a black box, something to be used but rarely understood. This article provides a short overview of the network stack software and how it works.

TCP/IP in a Nutshell

The Internet is built upon a small number of networking protocols which, when taken together, implement all that is necessary for one computer to talk to another. All these network protocols are implemented within the operating system's kernel. The reason network protocols are implemented within the kernel is that they are a resource that is shared among all of the users of a system, whether those users are humans or programs. Figure 1 shows some of the network protocols that implement the current Internet. We see that the protocols are layered and that they form a stack, hence the term Network Stack. Network protocols are thought of as layers because a system as complex as the Internet is built up of several, cooperating protocols, each of which has specific responsibilities and is dependent upon the layers below it to provide services necessary for the sum of the parts to implement the whole. Our figure shows three protocols, Address Resolution (ARP), Internet (IP) and Transmission Control (TCP).

For two computers to successfully talk to each other over an

Fig. 1 TCP/IP Model



Internetwork, three things must take place. The first is that information from one computer must be broken down into small enough pieces, which we call packets, to be transmitted between the source and destination of the communication. When you click on a link on the web and request a page, that page of data doesn't get transmitted as one large image of a cat; that image has to be broken down into pieces, often about 512 bytes in size; each piece must be sent across the network, and the destination computer must put all the pieces back together. The Transmission Control Protocol (TCP) is the protocol and layer that is responsible for the process of breaking down an image into packets, and giving each packet a sequence number, so that the destination computer can reassemble the packets it receives into the original image.

Once data is broken down into packets, they must be transported between the source and destination computers. The Internet Protocol (IP) is responsible for getting each individual packet between the source and destination. Every IP packet has both a source and destination address that indicates the ultimate endpoints of communication, but the Internet is a store-and-forward, packet-switching network, which means that most endpoints are not directly connected to the same local network. For an IP packet to reach its destination, it must be passed over a series of hops between one or more routers, before it reaches its final destination. The IP layer in the operating system kernel is responsible for finding the next hop along the path between the source and destination endpoints.

The Internet Protocols are abstract enough that various types of networking hardware can pass these packets without knowing anything about what is contained within the packets. Whether a packet goes over a wired or wireless link is unimportant to IP, but at some point, every computer or router has to contact the next hop along the path to the ultimate destination. For Ethernet-based networks, the Address Resolution Protocol (ARP) is responsible for finding the hardware address for the next hop along the path in the network. Every piece of Ethernet hardware has a 6-byte source and destination address, and it is ARP's job to translate a local IP address, usually of the next hop router, into a hardware address that the router is listening to.

Most programmers interact with the TCP/IP stack via the `sockets(2)` set of system calls. One of the key innovations of sockets is that it provides the programmer with an API that looks very much like a simple file access where all of the standard APIs, such as `read(2)` and `write(2)` work in the same way for network communication as they do for local file access. The Socket code can be thought of as a layer on top of the TCP/IP protocols that crossed the User/Kernel boundary to give programs access to network communication.

Summing up, the TCP/IP protocol suite running in a kernel on top of a wired Ethernet must: break a data stream into packets (TCP), address those packets so that they can move between a source and destination (IP), and finally figure out which piece of hardware is the next hop along the path between the source and destination (ARP). The sockets API ties the network stack back into user space so that programs can have access to the networking code. All of these layers are implemented in the FreeBSD kernel,

The Network Stack

Our goal here is not a full read through of the source code, which is well beyond the scope of this article, but, instead, to give you, the reader, an idea of how these pieces fit together and how you might even start learning how the code works.

The FreeBSD Network Stack is implemented in a set of C files contained in the operating system's `sys/` directory. The TCP and IP protocols are contained within `sys/netinet/`, and the ARP protocol, as well as much of the support for various link layers, such as Ethernet, are kept in `sys/net/`. The sockets API is mostly contained in the `sys/kern/` directory, along with much of the generic kernel infrastructure.

As the network stack is implemented as a set of layers, we must be concerned not only with the responsibilities of each layer, but also the form and function of how data is passed between them. Looking down from the top of the stack, a program hands a set of bytes of arbitrary length down into the network stack via the sockets API.

Each endpoint of communication for a program on a system is represented by a socket. The socket structure contains a great deal of metadata about the data it handles, as well as two queues for data—one for inbound communication, and one for outbound, which we call the receive and send socket buffers respectively.

All memory in the network stack is kept in a single, unified, data type called an `mbuf`, short for memory buffer. A set of `mbufs` can be chained together via a forward pointer, and this is how large areas of memory are broken down into packets that can be properly handled by the lower layers of the network stack. The `mbuf` system is a kernel private memory pool which user programs are never exposed to.

A `write(2)` call on a socket takes an area of memory and, via the system mechanism, makes that data available in the kernel, placing it into a socket buffer. Each socket buffer maintains a list of buffers that contain the data that's to be transmitted or that is in the process of being received. Once the data is contained in the socket buffers, the TCP machinery is invoked in the kernel and the `mbufs` contained in the socket buffer are updated and modified to be TCP segments, which are then further broken down by the IP protocol machinery and finally transmitted by the kernel via an Ethernet device driver.

Receiving data in the network stack is more complicated than reading from a file on a local system because network data can arrive at any time. A web server does not know to call `read(2)` before a web browser contacts it, and so the FreeBSD kernel must be constantly waiting for data and ready to create new communication endpoints as they arrive. When a client contacts a server, a TCP packet with a special flag (`SYN`) is received by the kernel, and the kernel then attempts to set up all of the state required for communication with the new endpoint. Only after the kernel has satisfied itself that it can communicate with the new endpoint does it alert a user space program that there is a new incoming connection, which the user space program can now `accept(2)` and then begin to `read(2)` data.

Look but Don't Touch (DTrace and the Network Stack)

How can we look at the network stack without trying to read the entire source code all at once? Using the built-in DTrace tracing system on FreeBSD, we can actually see each layer in action while running some simple test programs. As DTrace is completely safe to use on a running system, we can start to explore the network stack without having to modify and recompile the kernel code. For those not familiar with DTrace, you might want to start with the tutorial at https://wiki.freebsd.org/DTrace/Tutorial. Since this article presents a set of worked examples, it's easy enough to run these commands on a system of your own without knowing all the ins and outs of DTrace.

For simplicity we'll start with looking at an outgoing network connection. The `curl` command can be used to retrieve a single web page, and so we'll use www.google.com as our example.

As `curl` does not encrypt its data, unlike `ssh`, it is an excellent test tool with which we can look at the network stack.

Figure 2 shows how we can manually retrieve the base page from Google's website. The web page presented by Google is deceptively simple when seen in a web browser, but they embed a lot of code in their base page and so the output from the `GET` command runs to a few pages in a standard terminal. We're not interested in the output; we're interested in what happens when we initiate the communication.

Starting from the socket layer, we can see how `curl` begins communicating with Google by looking for calls to the `socket(2)` system call. All network communication requires a socket to be created. The socket call as seen by DTrace in Figure 3 doesn't show much of interest, and that's because all the `socket(2)` call does is set up the program's local endpoint for communication. To catch `curl` communicating with Google, we need to look instead at the `connect(2)` call. From a program's standpoint, `connect(2)` is the actual beginning of communication and is also the routine that will start the network stack's machinery which we can then observe. The `connect(2)` system call eventually leads, in the kernel, to TCP's connect routine being called, which can be seen using DTrace's `tcp:::connect-request` tracepoint. The connect-request tracepoint has quite a lot of information

Fig. 2

```
curl www.google.com

Lots of data from Google...
```

Fig. 3

```
▲  devbox ~  sudo dtrace -n 'syscall:freebsd:socket:entry /arg0==AF_INET/ {}'
dtrace: description 'syscall:freebsd:socket:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  1  60610                     socket:entry
```

Fig. 4

```
sudo dtrace -n 'tcp:::connect-request { print(args[3]->tcps_raddr); }'
dtrace: description 'tcp:::connect-request ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0  58881              none:connect-request string "172.217.8.4"
```

about the connection being attempted but for now we simply wish to see where the connection is going.

Figure 4 shows the output of a DTrace one-liner that catches `curl` in the act of contacting Google. We specify the connect-request tracepoint so that we can see our source machine trying to contact the destination, and we print the `tcps_raddr` (remote address) to see from what IP address Google is currently willing to communicate with us. If you run the `curl` command three times, you will see three lines of output, one for each connection.

Connecting the source system to the destination system requires the execution of the TCP state machine. Using the `/usr/share/dtrace/tcptrack`, we can see all of these state changes when contacting Google and retrieving its home page. Figure 5 shows the entire set of state transitions that the TCP layer moves through in order to set up a connection, retrieve data, and then close the connection and clean up after itself. Each socket starts out in a closed state (`state-closed`) and waits there until communication is initiated. When our source connects to the destination, it sends a special packet marked with a `SYN` flag, which moves the state machine into the `state-syn-sent`. Our socket will remain in this state until the destination replies and continues to set up the connection, indicated by state-established. The TCP state machine remains in the established state until one or the other side of the connection wishes to close it. When the connection is closed, the state machine moves through various states, all shown on the last three lines of Figure 5. A fuller discussion of the TCP state machine can be found in *The Design and Implementation of the FreeBSD Operating System*, but for our purposes, once the state transitions to `state-time-wait`, we are satisfied that this connection is closed.

Fig. 5

```
sudo ./tcptrack
State changed from state-closed                state-syn-sent
State changed from state-syn-sent              state-established
Connection established to 216.58.194.164:80 from 172.20.10.7:22045
State changed from state-established           state-fin-wait-1
State changed from state-fin-wait-1            state-fin-wait-2
State changed from state-fin-wait-2            state-time-wait
```

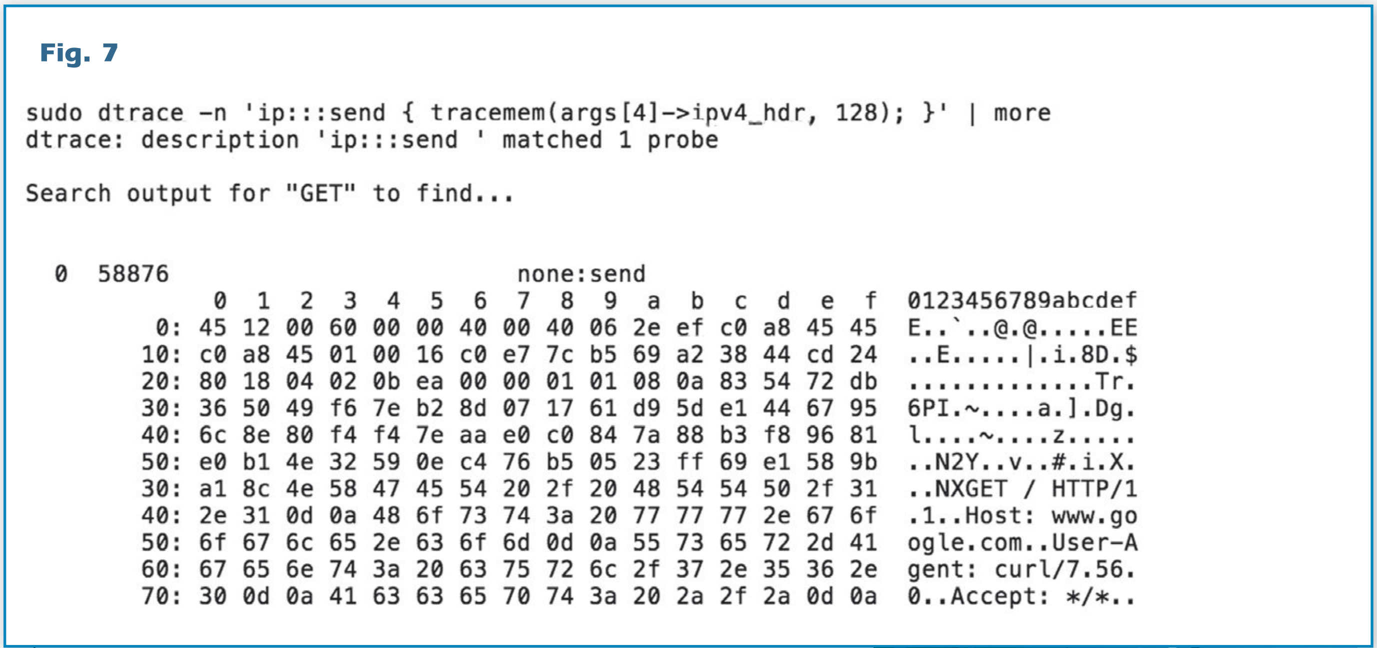
Fig. 6

```
sudo dtrace -n 'tcp:::send { tracemem(args[4]->tcp_hdr, 128); }' | more

Search output for "GET" to find...

0  58883                        none:send
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
         0: 86 f2 00 50 46 df a5 bc 67 fa 89 b2 80 18 04 0b  ...PF...g.......
        10: 51 6f 00 00 01 01 08 0a 00 43 d1 bd 5f b9 ba 6e  Qo.......C.._..n
        20: 47 45 54 20 2f 20 48 54 54 50 2f 31 2e 31 0d 0a  GET / HTTP/1.1..
        30: 48 6f 73 74 3a 20 77 77 77 2e 67 6f 6f 67 6c 65  Host: www.google
        40: 2e 63 6f 6d 0d 0a 55 73 65 72 2d 41 67 65 6e 74  .com..User-Agent
        50: 3a 20 63 75 72 6c 2f 37 2e 35 36 2e 30 0d 0a 41  : curl/7.56.0..A
        60: 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a 0d 0a 2f 97  ccept: */*..../.
        70: 4d 1a af 10 c3 50 53 ec c2 11 de 45 00 00 00 00  M....PS....E... . 1
```

While connection setup and teardown is a complicated affair, it actually doesn't move any of our data between systems. In order for us to see how `curl` communicates with Google, we can use DTrace's TCP send tracepoint. Figure 6 has a DTrace one-liner that will show all of the data being sent and received via TCP. If we were to try this test with `ssh(1)` or an HTTPS-enabled web server, we would not be able to find the plain text because the data would be encrypted before the TCP layer would see it, but with `curl` we get to see the plain text. In our example, we can see both the raw bytes in tabular form as well as the ASCII representation of the communication, and we can clearly see the `GET` command issued to the Google server. That's the same `GET` command that is passed down via a `write(2)` call into the network stack, which is then put into a socket buffer, handed to TCP, chopped up into packets, and finally transmitted via IP to the server.

In order to clearly see how the TCP and IP layers interact, we can compare the output of Figure 7 to Figure 6. Figure 6 was at the TCP layer and therefore only had TCP information, such as the packet's source and destination ports, sequence number, etc. Figure 7 is at the IP layer, one layer lower, and, as such, we

Fig. 7



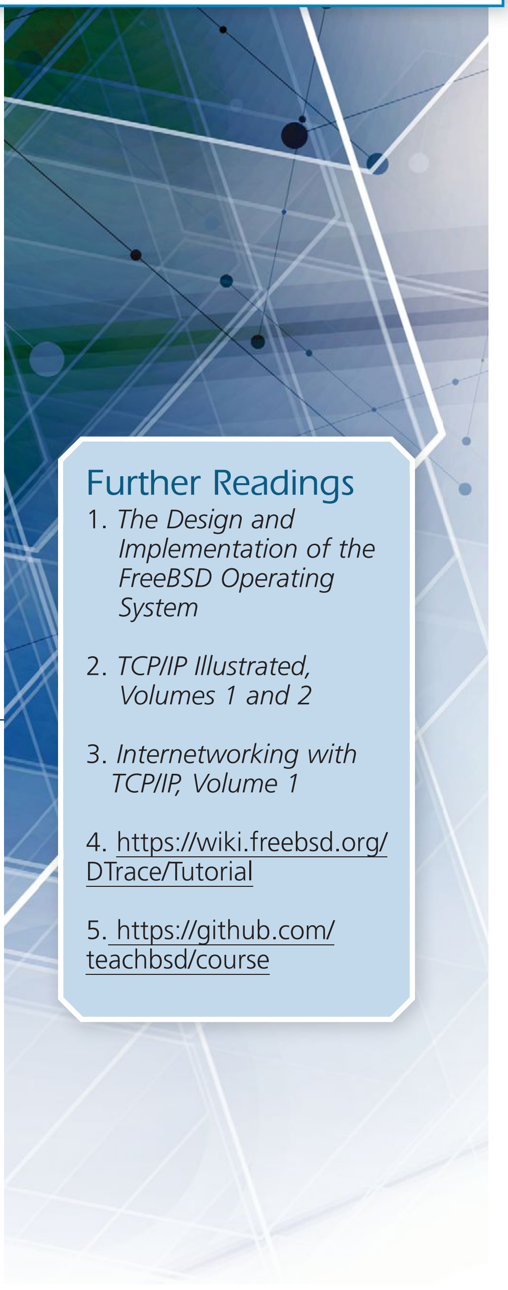
```
sudo dtrace -n 'ip:::send { tracemem(args[4]->ipv4_hdr, 128); }' | more
dtrace: description 'ip:::send ' matched 1 probe

Search output for "GET" to find...


  0  58876                        none:send
             0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f  0123456789abcdef
         0: 45 12 00 60 00 00 40 00 40 06 2e ef c0 a8 45 45  E..`..@.@.....EE
        10: c0 a8 45 01 00 16 c0 e7 7c b5 69 a2 38 44 cd 24  ..E.....|.i.8D.$
        20: 80 18 04 02 0b ea 00 00 01 01 08 0a 83 54 72 db  ............Tr.
        30: 36 50 49 f6 7e b2 8d 07 17 61 d9 5d e1 44 67 95  6PI.~....a.].Dg.
        40: 6c 8e 80 f4 f4 7e aa e0 c0 84 7a 88 b3 f8 96 81  l....~....z.....
        50: e0 b1 4e 32 59 0e c4 76 b5 05 23 ff 69 e1 58 9b  ..N2Y..v..#.i.X.
        30: a1 8c 4e 58 47 45 54 20 2f 20 48 54 54 50 2f 31  ..NXGET / HTTP/1
        40: 2e 31 0d 0a 48 6f 73 74 3a 20 77 77 77 2e 67 6f  .1..Host: www.go
        50: 6f 67 6c 65 2e 63 6f 6d 0d 0a 55 73 65 72 2d 41  ogle.com..User-A
        60: 67 65 6e 74 3a 20 63 75 72 6c 2f 37 2e 35 36 2e  gent: curl/7.56.
        70: 30 0d 0a 41 63 63 65 70 74 3a 20 2a 2f 2a 0d 0a  0..Accept: */*..
```

see that the GET command has moved a considerable way further down into the data stream. The information required by the IP layer, such as the destination and source network address, is responsible for this displacement of the GET command.

Conclusions

Networking is a complicated topic and its implementation in the operating system kernel is covered at various levels in some of the Further Readings (1, 2 3). Understanding how the network stack works requires remembering that the network stack is broken down into modules that roughly match the protocols that are being implemented. The Internet Protocols are in the IP layer, and the Transmission Control Protocol is contained in the TCP layer, and these layers interact with each other through a small number of well-defined kernel APIs. The DTrace system on FreeBSD is the perfect tool for those who wish to start exploring the network stack. Various DTrace-related tutorials can be found in the Further Readings section at right (4, 5). ●

GEORGE V. NEVILLE-NEIL works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, networking and time protocols. He is the coauthor with Marshall Kirk McKusick and Robert N. M. Watson of *The Design and Implementation of the FreeBSD Operating System*. For over 10 years he has been the columnist better known as Kode Vicious. He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler and currently lives in New York City.



Further Readings

1. *The Design and Implementation of the FreeBSD Operating System*

2. *TCP/IP Illustrated, Volumes 1 and 2*

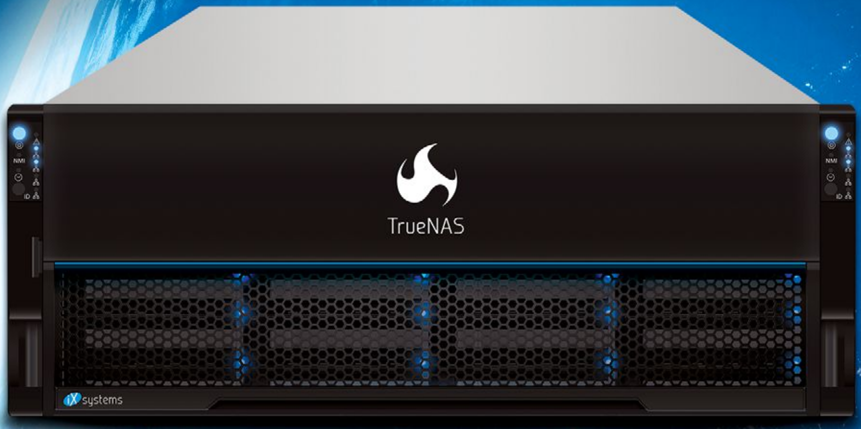
3. *Internetworking with TCP/IP, Volume 1*

4. https://wiki.freebsd.org/DTrace/Tutorial

5. https://github.com/teachbsd/course

Groundbreaking TrueNAS® M Series

DISRUPTING STORAGE AT WARP SPEED



LOWEST TCO SHARED STORAGE

Intel® Xeon® Scalable Family Processors

RESILIENT

- Self-healing Data
- Continuous Operation
- Easy Replication

WARP FACTOR IX

- Faster than SSD-based Hybrid Storage Arrays
- Flash-Turbocharged Data Access

EXPANDABLE

- Scales to 10PB using HGST Drives
- 10Gb/s-100Gb/s per NAS
- Non-disruptive Upgrades

COMPATIBLE

- Citrix, Veeam, & VMware Certified
- Unifies File, Block, & S3 Data
- Supports Leading Cloud Providers

Visit iXsystems.com/TrueNAS or call (855) GREP-4-iX today!

HGST a Western Digital brand

iX systems™

SEE TEXT ONLY

TCP STACK VALIDATION AT NETFLIX

By Jonathan **Looney**

From the time I started writing TCP code, there has always been a great deal of fear around changes to the TCP stack. This comes from multiple sources, but is rooted in the difficulty of thoroughly testing TCP changes prior to deploying them. This is a challenge Netflix faced as it began investing significant time and energy in optimizing the FreeBSD TCP stack used on its Open Connect Appliances (OCAs), which are the servers that deliver video traffic to Netflix streaming clients. In this article, we will describe the way Netflix removed the fear from TCP development and actually gained the ability to confidently test its changes.

The Challenge

There are many reasons why there has traditionally been fear around making changes to the TCP stack.

One challenge is the code complexity. The TCP stack is quite mature. That maturity brings value, because the TCP stack has shown itself to be reliable over a long period of time (and includes myriad bug fixes to enhance that reliability). But that maturity also brings with it complexity: there is legacy code that may not have been cleaned up, special cases that have been added on over the years, and various branches that can be somewhat obscure. That complexity means it is quite easy to break something unintentionally.

To make the task even more challenging, once you have written your code, it is hard to validate whether things are truly "better" or "worse." (And, in fact, something might be "better" in some way while being "worse" in another way.) For example, let's assume you are trying to fix a problem where the TCP stack will retransmit TCP segments unnecessarily. You may fix that bug (lowering the number of unnecessary retransmissions) but accidentally reduce goodput (the effective data throughput rate to the client application) because you are no longer retransmitting as aggressively. However, given the very wide variation in client behavior and network conditions that exist on the Internet, it is hard to test even a reasonable fraction of the possible cases in a lab.

It is also easy to introduce truly catastrophic bugs without noticing. For example, it is not unheard of for a TCP developer to make a seemingly-innocent change and to find out (usually, some time later) that a timer is no longer arming in a corner-case situation and some TCP connections are becoming stuck. Likewise, we have seen cases where a TCP stack fails to send data when it should.

And, to make this all much more complicated, mistakes are hard to fix. After all, if you break TCP connectivity, you may not even be able to copy a fixed kernel to the device.

For all of these reasons, it is both important to thoroughly test TCP changes and also very hard to do.

The Tools

At Netflix, we use various tools to help us validate TCP changes. During and immediately after development, we may use various unit tests or Packet Drill tests to validate that we are seeing the correct behavior. However, once we move on to lab testing or field testing, we fundamentally use three tools to help us validate TCP changes, client-reported and server-reported metrics, modular TCP stacks, and the Blackbox Recorder.

Metrics

Once a new TCP stack has reached the point of being a candidate for deployment, we begin to have clients use it. Both our clients and our servers gather detailed statistics about their TCP sessions. This includes information such as goodput, RTT, and the number of retransmissions. The clients also tell us how quickly they got the first chunk of data, how steady the data stream was, and whether they encountered any disruptions to the data stream during playback. (And, of course, if the performance is too far below what the clients are expecting, they will attempt to switch their traffic to an OCA outside of the test. This keeps us from causing too much of a negative impact to client traffic.)

Being able to see metrics from both sides of the connection is a great help in understanding the way the code is functioning. Further, Netflix has a robust data analysis pipeline that allows us to retrieve detailed reports comparing TCP connections that used different TCP stacks and to determine whether the differences are statistically significant. By looking at this data, we are able to detect fairly small differences between two TCP stacks.

However, this still leaves an obvious need: you must be able to compare two TCP stacks in a reliable way. For that, we use modular TCP stacks.

Modular TCP Stacks

Netflix uses the modular TCP stack functionality to run multiple TCP stacks on a single server. This has a number of benefits. First, we can choose to use different TCP stacks for different applications. Second, we can thoroughly test new TCP stacks side-by-side with older TCP stacks. Finally, we can (at least, in theory) deploy fixed versions of TCP stacks without requiring an upgrade to the underlying operating system. (We don’t currently regularly make use of on-the-fly TCP stack changes, but we are currently testing the infrastructure to enable that.)

To understand how modular TCP stacks benefit Netflix, it may help to explain the way we now do TCP development at Netflix.

We do our development on a development copy of the TCP code. When we ship a new candidate TCP stack, we copy that code to its own directory and give it a version that is incorporated into its name. Once copied, we try to only update that code to deal with necessary API changes. The modular stack infrastructure in FreeBSD allows us to compile the same code, but use different stack names. It conducts symbol mangling to prevent symbol conflicts between different copies of the same code and lets us install the different versions with different stack names. This lets us literally copy our development code into a new TCP module directory, change one or two Makefile variables, and instantly have a new version of the TCP stack.

(By analogy, you can think of these as being similar to FreeBSD releases. We develop on stable/11 and then copy to the releng/11.0 branch. But we will keep developing on stable/11 and eventually copy it to the releng/11.1 branch. At Netflix, we treat our TCP stacks similarly to this paradigm.)

This ability to compile the code unmodified but with a new TCP stack name is a subtle, but important, feature. This lets us directly track code changes between versions without getting distracted by extraneous, non-functional changes that are there solely to support the module renaming.

When Netflix has a candidate TCP stack, we will deploy the old and new TCP stacks side-by-side on the same OCA. We will then have some number of clients use one or the other of these stacks on that OCA and report their metrics. We can then gather the statistical data and compare the performance of the two stacks.

For the test, it is important that we run the old and new stacks side-by-side on the same OCA. This eliminates many variables that could influence the results. The underlying hardware, operating system, and operating environment should impact the two TCP stacks in exactly the same way. By running the two stacks side-by-side on the same OCA, we are able to focus on just the differences caused by the way the TCP stacks—themselves—perform.

The modular TCP stack functionality also provides a smooth transition to the deployment of the new TCP stack. Once we have validated the new TCP stack on a small-scale test, we move to validate it with gradually larger tests. In our final test, we might have the new TCP stack handle 50% of all Netflix client traffic globally and validate that the stack still performs as expected. If it does, we can switch all clients to use the new stack by default. (And, because the stack can be selected at runtime, it doesn’t even require a reboot of the OCAs.)

Using the modular TCP stack functionality, we can also choose the best TCP stack for each application and client. For example, we might find that the new TCP stack performs noticeably worse for a subset of clients. We can set those clients to use the old TCP stack while we investigate that further while we have the rest of the clients benefit from the new TCP stack’s improvements. Likewise, we can conduct this testing separately for each application running on our OCAs and change those applications to use the new TCP stack on their own schedule. This lets us ensure we are using the TCP stack that we've validated best serves the needs of our clients when using each application on our OCAs.

The modular TCP stack functionality also provides us with protection against TCP stack bugs. We have occasionally found very serious bugs in the new TCP stacks. We are able to dynamically unload those modules without interrupting our services. The OCA (and our clients) continue to function using the old TCP stacks installed on the OCA.

Once we fix the bug, we can deploy a new TCP module, load it, and begin testing it. Again, this doesn't require a reboot and is fairly seamless. At the moment, we only regularly use this in development, but we are now testing the tooling we've written to enable us to automate dynamic TCP module deployment across our network of OCAs.

Through the combination of these capabilities, we can have much more confidence deploying our new TCP stacks. We are able to easily create named "release candidate" versions of our TCP stacks, deploy those on OCAs, test them side-by-side with the existing stacks, and—if necessary—recover from a serious TCP stack bug and iterate with new versions that fix bugs in the TCP stack. This is a drastically different picture than the culture of fear described earlier. This is a healthy environment in which to do confident TCP stack development and it is enabled by the ability to easily build, deploy, and use modular TCP stacks.

But, the testing up to this point has focused heavily on examining metrics maintained by the client and server. There is one more aspect to our testing which is worth noting, and that is provided by the TCP Blackbox Recorder.

TCP Blackbox Recorder

The TCP Blackbox Recorder provides a data stream of events from TCP connections. It is named the "Blackbox Recorder" after the flight data recorders (colloquially known as "blackbox recorders") carried on commercial airliners. As initially conceived, the Blackbox Recorder would log a stream of events that occurred on a TCP connection to a ring buffer associated with that connection. If either the user-space application or the kernel notices that something has gone "wrong" with the connection, it can dump out the contents of the ring buffer for later analysis. (And, in the case of kernel panics, a developer can pull data from the ring buffer to see the sequence of events leading up to the crash.)

The events each contain a record of the internal state of the TCP connection so a developer can track how the internal state changed between events. By tracking both internal events and the internal state of the TCP connection, a developer can get a good record of why a connection behaved in the way it did. In fact, this internal information can produce valuable insights that are hidden from analysis tools that only track what was sent and received.

That functionality is very useful and we've used it to debug problems at Netflix. However, the Blackbox Recorder also has another mode of operation that can be useful in finding problems. The Blackbox Recorder allows us to select a percentage of TCP connections for "continuous" logging, where it tries to export all the events from the TCP connection to user space for later analysis. This data can be useful in understanding exactly what has occurred on TCP connections—even connections that we think look "normal."

During development, we may manually scan a sampling of these records to ensure the behavior matches our expectations. We will pay particular attention to both areas we think should have changed due to our code enhancements and also areas we fear might have been accidentally impacted by our code changes. However, there is a limit to how much we can detect through manual review.

We also have an automated tool that scans the Blackbox records to validate that each session operates in keeping with some basic assumptions. For example, we check that we are sending data at least once every second or two when we have no reason to pause (there is window space available and data to send). This lets us validate that timers are working as expected. We also check that we are not exceeding the congestion window and peer's receive window for the connection. This lets us validate that we are not sending data when we should not be doing so. These types of very basic sanity checks are valuable in finding corner-case bugs that escape our lab testing, but are revealed once we start wider testing with real clients.

When this system finds an error, it posts an alert for the TCP development team. They can retrieve the trace and try to determine why the error occurred. And, thanks to the state information logged with the events, it is usually fairly easy to isolate the problem (or, at least, the problematic area of the code).

An Example

It might help to give an example of how this works using a contrived example that is an amalgamation of experiences we have had using this infrastructure and methodology.

For this example, let's assume the current version of the TCP stack we are using is rack_11. The development version is rack_12 but has not changed since rack_11 was created. (In other words, rack_11 and rack_12 are using the same code.) We want to fix a bug with unnecessarily high retransmissions in rack_11.

We make our code changes to the development version and test it. We think it fixes the problem. We now deploy both rack_11 and rack_12 to an OCA and run a test with a small number of clients. The metrics look good, so we expand the test to cover a few more OCAs and additional clients. At this point, we notice that retransmissions have dropped (which was expected and is "good"). But, we also notice that goodput has dropped (which was not expected and is "bad"). We gather Blackbox traces and find that we accidentally used a <= comparison where we should have used a < comparison. This caused us to not retransmit in some cases where we should have retransmitted.

We fix this bug in the development version and test it again. We think it fixes the new problem (while also still fixing the original problem). We again deploy both rack_11 and rack_12 to an OCA and run a test with a small number of clients. The metrics look good, so we expand the test to cover a few more OCAs and additional clients. The metrics still look good, so we commit the code.

Eventually, rack_12 becomes the release candidate and rack_13 becomes the new development version. At this point, we deploy rack_12 to a larger set of OCAs. The metrics still look good. However, at this point, the Blackbox analysis program alerts that we are failing to send data in a small number of cases. We realize that we have accidentally failed to reset the retransmit timer in a particular corner case.

We fix this new bug in rack_13 (the new development version) and test it again. We deploy both rack_11 and rack_13 to an OCA and run a test with a small number of clients. We then request that our release engineer merge our commit to rack_12. For the sake of our example, we'll say he agrees, so rack_12 now has the new bug fix. We deploy this to the larger set of OCAs and continue our release testing. The metrics still look good, and there are no more alerts from the Blackbox analysis program.

Now, we deploy rack_12 to all OCAs in the Netflix network. We test with a small percentage of clients. The metrics look good, so we proceed with testing against 50% of all Netflix clients. At this point, we notice that the metrics for one client operating system are poor, while the metrics for all others are the same or better. We decide to direct all Netflix clients to use the new TCP stack, but we make an exception for the one operating system with worse metrics and tell the clients using that operating system to use the old TCP stack.

We then begin to gather focused testing and debugging data for that one client type in an effort to understand why it performs more poorly with the new TCP stack code than with the old code. Hopefully, we are able to fix the bug in rack_13. At that point, we start the testing process again.

In reality, this process sometimes takes longer than we would like. For example, we have deployed release candidates across all OCAs in the Netflix network, only to abandon the release candidate in a late stage of testing due to a subtle metric change that escaped detection during earlier, more narrow testing phases. But, even if it sometimes takes longer than we would like, we are happy with the result this thorough testing process produces: the ability to confidently upgrade the code in our TCP stacks.

TCP Development with Confidence

Given all these tools, we are able to conduct TCP development with a greater degree of confidence that we are making things better. We know that we will not write bug-free code. We know that we will not catch all the problems in a lab. But we can slowly and carefully deploy TCP stack changes and test them in a way that ensures we can test TCP stack changes without having our results influenced by differences in the underlying hardware or operating system. And we can validate that the new TCP stack behaves correctly both by looking at server-side and client-side metrics and also by examining traces of the internal operation of the TCP stacks.

The combination of these things has changed the TCP development paradigm from one constrained by fear to one where we have great freedom to innovate, confident that we have the right tools to innovate responsibly.

Note: Much of the code (including the RACK TCP stack) described in this article is already available in FreeBSD 12. A few things (such as enhanced server-side stats and the user-space Blackbox analysis tools) are still in the process of being upstreamed, but should land "soon." ●

Jonathan Looney manages a development team at Netflix responsible for maintaining the operating system that runs on the OCAs. He is a FreeBSD committer active in the transport protocols area.

SEE
TEXT
ONLY

A Quick Tour OF SDN USING MININET

I've been running both FreeBSD and Linux uninterrupted since the mid-1990s. I've run many different Linux distributions over the years, most recently focusing on CentOS. I also have significant experience with Mac OS X and NetBSD and have experimented with many other BSD platforms. As a staunch agnostic with a firm belief in the value of open standards, I like to remain familiar with all the options in the POSIX world so I'm always prepared to choose the best tool for the job.

By Ayaka **Koshibe**

In the most basic sense, software-defined networking (SDN) can be thought of as an approach to building a network that can be managed as if it were one logical entity. An SDN-based network is typically built from programmable whitebox and software switches, and is managed from control applications that use their global view of the network to coordinate the switches to act as one. The result can look fairly unfamiliar to those used to "classic" networks that are configured on a per-device basis and the behavior of which is determined by distributed network protocols. Network emulators can be useful tools for gaining better insight into how these networks behave and are put together.

Mininet

Mininet is a fairly well-known emulator for SDN-based networks that was popularized by its ties to OpenFlow, a network control protocol from the dawn of SDN. It was also recently added to the ports collection. Using that as an occasion, here is a quick tour of Mininet in the form of an SDN primer.

Before we begin: Mininet depends on VIMAGE for emulating network hosts, so readers wishing to follow along will need a host with VIMAGE support. The ported version of Mininet also doesn't support the full set of features of the original and is very much a work-in-progress. It is also heavy-handed with cleanup, so it is best not run on machines used for hosting other jails or Open vSwitch instances.

Installation

Mininet can be installed like any other application: with `pkg(8)` as 'py27-mininet', or from the ports tree as 'net/mininet'.

The *mn* Command

The Mininet version of a "Hello world" is a tiny network launched with the *mn* command:

```
# mn --controller=ryu
*** Creating network
...
*** Starting CLI:
mininet>
```

This creates a network with two hosts connected through a switch controlled to act as a learning switch. It also launches a CLI for interacting with the network. For example, ***links*** shows all of the links in the network:

```
mininet> links
h1-eth0<->s1-eth1 (OK OK)
h2-eth0<->s1-eth2 (OK OK)
```

And ***dump*** shows information about the nodes in the network:

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=8410>
<Host h2: h2-eth0:10.0.0.2 pid=8414>
<OVSSwitch s1: lo0:127.0.0.1,s1-eth1:None,s1-eth2:None pid=8420>
<Ryu c0: 127.0.0.1:6653 pid=8401>
```

Either ? or ***help*** will show all available commands.

Inspecting Control Traffic

The output of ***dump*** shows that each node has a name, one or more ports, and a PID of the bash process that represents it. It also shows that the hosts—actually vnet jails—are at 10.0.0.1 and 10.0.0.2 in this network, and the controller, Ryu, is listening for switches on port 6653. Ryu uses OpenFlow to program the switches that connect on this port. A typical way of troubleshooting OpenFlow switches and controllers is to inspect the control messages on this channel. We can try this by running tcpdump (or another packet analyzer) in another terminal:

```
# tcpdump -i lo0 port 6653
```

We should be able to see the keepalive `ECHO_REQUEST` and `ECHO_REPLY` messages sent between `s1` and `c0`. Next, ping one host from the other. The CLI interprets any commands after a host's name as bash commands to be run from that host:

```
mininet> h1 ping -c1 h2
```

A host's name is translated by the CLI into its corresponding IP address.
Alternatively, the ***pingall*** CLI command can be used to ping between all pairs of hosts:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

In either case, we should see the switch (`localhost.<high port>`) sending `PACKET_IN` messages to the controller (`localhost.6653`), and `PACKET_OUT` and `FLOW_MOD` messages sent back by the controller in response. Switches use `PACKET_IN`s to send packets that it doesn't understand how to process to the controller, in this case, the ARP and ICMP messages. Controllers use `PACKET_OUT`s to instruct a switch to output a particular packet (the one sent in the `PACKET_IN`, so that it is not "lost"), and `FLOW_MOD`s to modify how a switch handles different types of traffic. No new `PACKET_IN`s should be generated by `s1` in response to another ping until the modifications expire from disuse.
A ctrl-D or the ***exit*** command will exit the CLI and tear the network down.

Experimenting with Controllers

The controller's role in the network can be directly demonstrated by running a network without one. This can be done by passing 'none' instead of 'ryu' to ***mn***'s *—controller* option. The hosts on this "headless" network should not be able to ping each other.
Another useful option is 'remote', which allows a network to use a controller running outside of Mininet's

control. Developers might test their controllers by pointing a Mininet network at them with this option. Assuming that a controller is running at 192.168.0.100 and listening on port 6633, the following will launch a network and connect the switch to it:

```
# mn --controller=remote,ip=192.168.0.100,port=6633
```

Creating Various Topologies

The *--topo* option is used to create various topologies with *mn*. The *linear* and *tree* topologies are useful for creating larger loop-free networks, whereas the *torus* topology is useful for testing a controller's loop-handling abilities. Topologies are parameterized so that their sizes can be specified. For example, to create a tree three levels high and fanout of two:

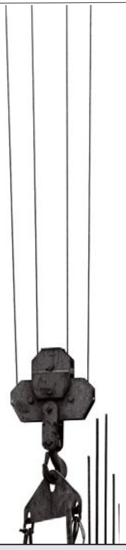
```
# mn --controller=ryu, topo=tree,3,2
```

The *torus* also takes two values, and *linear* takes one.

Scripting with Mininet

Mininet can also be used as a collection of Python libraries for scripting experiments. With the caveat that they are in their original forms (and will most likely not work on FreeBSD), the package includes several example scripts that demonstrate how to create custom topologies, network components, and experiments. As with other applications that come with examples, they should be found under `/usr/local/share/examples/mininet/`. But, as a small example, the following script defines a custom topology resembling mn's default topology, uses host `h1` to ping `h2`'s address, and exits:

```
from mininet.topo import Topo
from mininet.net import Mininet

class MinimalTopo (Topo):
  def build(self):
    h1 = self.addHost('h1', ip='192.168.0.1')
    h2 = self.addHost('h2', ip='192.168.0.2')
    s1 = self.addSwitch('s1')

    self.addLink(h1, s1)
    self.addLink(s1, h2)

net = Mininet(topo=MinimalTopo())
net.start()
h1 = net.getNodeByName('h1')
print(h1.cmd('ping -c1 192.168.0.2'))
net.stop()
```



Once saved, it can be run like any Python script:

```
# python example.py
```

Finding Out More

While only a subset of the upstream features are supported by this port, the main project maintains resources that should provide a better idea of how Mininet can be used. These can be found at:
https://github.com/mininet/mininet/wiki/Documentation

And the port itself is maintained at:
https://github.com/akoshibe/mininet

So this concludes our whirlwind tour of Mininet. Hopefully, it serves as a decent starting point for those interested in exploring the area of SDN. •

Ayaka Koshibe became involved in the area of SDN as a college student assisting in the deployment of infrastructure for the GENI OpenFlow campus trials. She currently works at Big Switch Networks as a member of the SDN controller platform team, and is also both maintainer and upstream for the Mininet port for FreeBSD and OpenBSD.

BSD CERTIFICATION GROUP
HAS JOINED WITH LPI.

ONE MORE STEP TOWARDS A

FREE AND OPEN
SOURCE WORLD

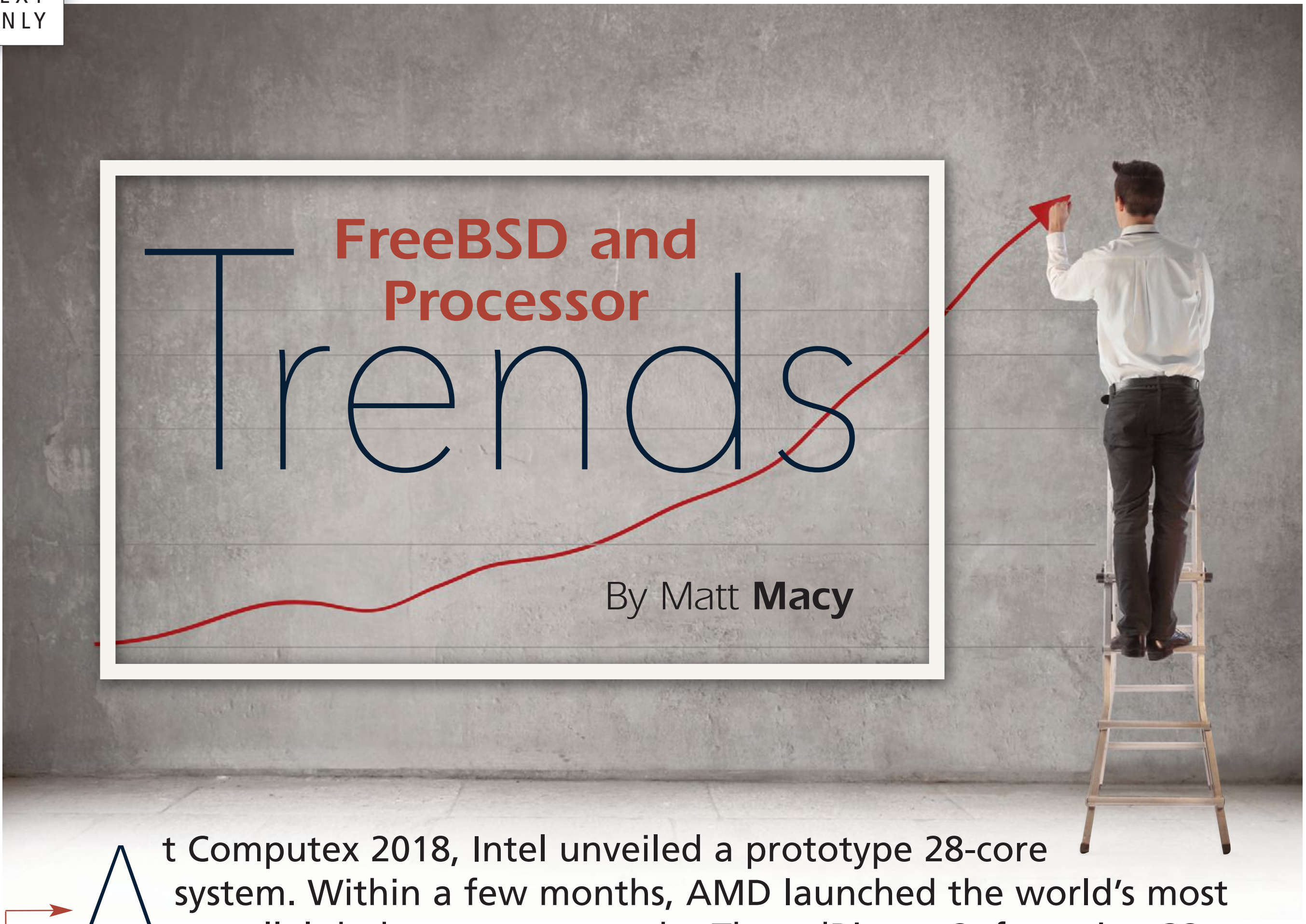
NOT TO MENTION, JOBS!

Now as a part of LPI, BSD certification will gain a new global reach. It will also benefit as it's relaunched in 2018 to fit into a broader program of free and open source professional credentials. You can help by participating in retooling the certification at lpi.org/bsd.

Linux
Professional
Institute

COMING IN 2018.
WATCH FOR UPDATES.

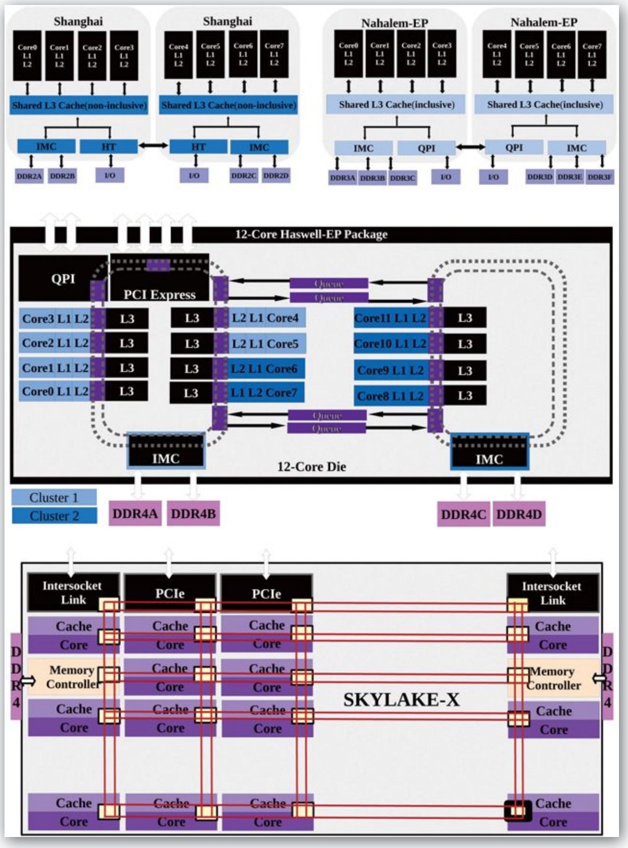
SEE TEXT ONLY



FreeBSD and Processor Trends

By Matt **Macy**

At Computex 2018, Intel unveiled a prototype 28-core system. Within a few months, AMD launched the world's most parallel desktop processor, the ThreadRipper 2, featuring 32 cores (64 hardware threads). AMD's EPYC2 is in the lab and rumored to be 64 cores (128 hardware threads), bringing 256 hardware threads to a commodity server dual socket system. Historically, FreeBSD has existed at the "knee" of the hardware commodity curve. In order to maintain its relevance in the server space, FreeBSD needs to keep pace with the latest processor developments.



Processor Evolution

As core count has increased, the designs have gotten steadily more complicated. AMD's Shanghai and Intel's Nehalem used a broadcast bus for handling cache coherence. Intel's Haswell later changed this to multiple rings on chip. And with Skylake, Intel has moved to a mesh **(see right)**.

AMD has taken a yield-centric focus to scaling up by reusing the same design across its product line. Each chip consists of two core complexes (CCX), and an EPYC package consists of four chips (often referred to as "chiplets") **(see lower right)**.

Defining Scalability

Scalability can be defined on a number of axes [Culler, 1999]:

- Problem-Constrained `strong scaling` - The user wants to use a larger machine to solve the same problem faster. As the number of processors available to complete a task increases, the extent to which the time completes the problem decreases:

$$\text{Speedup}_{PC}(n \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(n\text{processors})}$$

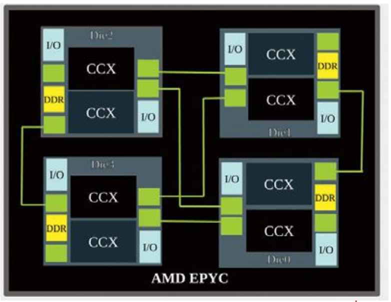
- Time-Constrained `weak scaling` - the time to execute a given workload remains constant; user wants to solve the largest problem possible. It is the degree to which the amount of work accomplished increases as the number of processors increases:

$$\text{Speedup}_{TC}(n \text{ processors}) = \frac{\text{Work}(n \text{ processors})}{\text{Work}(1 \text{ processor})}$$

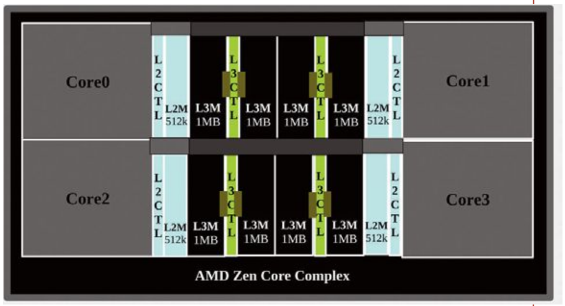
- Memory-Constrained - The user wants to solve the largest problem that will fit in memory.

$$\text{Speedup}_{MC}(p \text{ processors}) = \frac{\text{Work}(p \text{ processors})}{\text{Time}(p \text{ processors})} \times \frac{\text{Time}(1 \text{ processor})}{\text{Time}(1 \text{ processor})} = \frac{\text{Increase in Work}}{\text{Increase in Execution Time}}$$

In this article, scalability refers to time-constrained scalabilty ("weak scaling"), which will be characterized by the aggregate number of operations performed during benchmarks. Performance bottlenecks are application- and work-load specific. Therefore it is problematic to extrapolate actual application performance from these scalability measurements. Nonetheless, the OS impact on any given workload can be characterized as a combination of the average time per system call and the impact of scheduling decisions. System call overhead can be captured by simple microbenchmarks. Scheduling decisions are harder to measure but one can measure them to a limited degree by measuring workloads with varying scheduler restrictions (i.e., limiting the set of CPUs the scheduler can use) or by comparing single socket results with dual/multi-socket results.



It is important for the reader to understand that the purpose of microbenchmarks is not to measure workloads themselves. They are a means to observe the scaling of individual OS services to measure scalabilty in isolation. These measurements are only predictive of performance on real world workloads to the extent to which a workload uses the individual service being measured.



What Makes Scaling Difficult—Serialization and Scheduling

If n threads are attempting to perform an operation, serialization overhead can roughly be defined as the extent to which the throughput per thread declines from 1 to $1/n$, as each thread waits to acquire the same lock. `Scheduling overhead` is more difficult to define. In an ideal world a given thread would only ever run on one core; any other threads that it communicated with would be on the same "core complex" - sharing an L3 cache so that IPIs (inter processor interrupts—a facility to allow a cpu to interrupt other cpus) and cache coherency traffic would not have to traverse an interconnect and any misses could be refilled without going to memory. Unfortunately, in practice, this is impossible in the general case. CPUs are commonly oversubscribed and the scheduler cannot infer relationships between threads in different processes. The further away one cpu is from another, the poorer the performance for latency-sensitive operations such as networking and synchronous IPC. And the larger the distance between two cpus, the greater cost of refilling the caches when a thread is migrated.

The scaling challenges stem from three factors:

- memory latency
- limits to coherency traffic
- shared globally unique resources

To a large degree the scaling solutions are all a combination of:
- per cpu resources
- relaxing constraints
- distinguishing between existence guarantees and mutual exclusion

Memory latency and the bounds on coherency traffic are fundamental to the evolution of computer hardware over the last decade. What were once design artifacts seen only in high-end systems are now an important consideration even in consumer CPUs like AMDs ThreadRipper. The shared memory programming model is becoming an increasingly leaky abstraction. Cache coherence logic in processors provides the single-writer /multiple-reader `SWMR` guarantees that programmers are all accustomed to [Sorin, 2011]. However, at its limit, the observed performance is defined by the actual implementation of a distributed memory with all updates performed by message passing [Hacken, 2009], [Molka, 2015]. Today, message latency and bandwidth are dominant factors in observed performance.

Implementation issues impacted by the increasing number of hardware threads are:
- locking granularity
- using locks to provide existence guarantees
- using atomic references to provide existence guarantees
- poor cache locality between L3 caches or NUMA domains.

Locking Granularity

Locking granularity refers to how many operations are protected by a single lock. The "Big Kernel Lock" or "BKL" in Linux or "Giant" in FreeBSD initially encompassed the entire kernel in a single lock. This evolved into locks for individual subsystems, then individual data structures, and finally fields in data structures. Even with fine-grained locking, the case of a widely referenced global resource (memory, routing table entry, etc.) that can only be accessed one at a time occurs. In general, locking granularity in FreeBSD is already relatively fine-grained. Nonetheless, between FreeBSD 11 and FreeBSD 12 there are numerous examples of work done to reduce lock contention, either by increasing locking granularity, moving to per-cpu resources, or reducing the frequency with which global updates occur.

Locking for Existence Guarantees

One can use a lock to guarantee that entries in a system global or process global structure have not been freed while in use. One example in FreeBSD 11.x vs FreeBSD 12.x is how existence is guaranteed for connection state within the per protocol hash table. FreeBSD 11.x guarantees a thread that any connection found in the table is valid by requiring that all table readers do a shared (for read) acquisition of a per-table reader/writer lock. This allowed multiple simultaneous readers while preventing any table updates. Although conceptually straightforward, this comes at a substantial price and the guarantee is stronger than required. FreeBSD 12.x weakens the guarantee to provide that any connection found during a lookup had not been freed. Lookups are protected with epoch and updates are serialized with a mutex. Connection state lookup still returns the connection locked to guarantee existence past lookup. However, once the lock is acquired, lookup now checks that the connection has not had the `INP_FREED` flag set. If the flag is set, this indicated that connection is pending free. In this case, we drop the lock and return `NULL` as if no connection had been found. This change adds some additional complexity to readers, but in exchange we no longer require a global atomic for the rwlock [App. A] and updates can proceed in parallel with lookups (lookups no longer block on updates and vice versa). This change provided a 10–20x reduction in time spent in lookups on a loaded multi-socket server.

Atomic Refcounts for Existence Guarantees

Atomically updating a reference counter for an object peforms better than using a lock to serialize updates. Updates can proceed fully in parallel with ownership changes. Each new thread or object holding a pointer to the object increments the reference. When the reference is removed from the object or the thread's reference goes out of scope the reference is decremented. When the count goes to zero the referenced object is freed. Nonetheless it does not scale as the cost of coherency traffic rises. For an object frequently referenced by many threads the coherency traffic invalidating and migrating the cache line between L2 and L3 caches quickly becomes a bottleneck. There are two separate issues to address here:
- Is reference counting necessary here?
- Can anything be done to make reference counting cheaper?

Perhaps suprisingly, for stack local references, reference counting isn't actually necessary. SMR "Safe Memory Reclamation" techniques such as Epoch Based Reclamation [Fraser, 2004], Hazard Pointers [Michael, 2004; Hart, 2006], RCU [McKenney, 2011], scalable parallel sections [Wang, 2016], etc., can allow us to provide existence guarantees without any shared memory modifications. And reference counting can, in many cases, be made much cheaper.

Recent work in UDP expanded the scope of objects tied to the network stack's epoch structure. Epoch structure is now also used to guarantee existence of interface addresses. This now means that references to them that are stack local no longer need to update the object's refcount.

The observed reference count can safely be different from the "true" reference count if we can safely handle zero detection correctly. The different approaches to scalable reference counts rely on this insight. Although there are other approaches to this in the literature [Ellen, 2007], the ones I consider most interesting are Linux's percpu refcount [Corbet, 2013] and Refcache [Clements, 2013]. The former is a per-cpu counter that degrades to a traditional atomically updated reference count when the initial reference holder "kills" the perpcpu refcount. Its advantage is that it is simple and can be extremely lightweight provided that the life cycle of the object closely mirrors that of the initial reference holder. It does not work well if the object substantially outlives the initial owner. Refcache maintains a per cpu cache of reference updates and flushes them when there is conflict or at the end of an "epoch." In this case an "epoch" is 10 milliseconds. Zero detection is done by putting the object on a per-cpu "review" list when its global reference count reaches zero. The global reference count can be assumed to be the true reference count when it has remained at zero for two "epochs." Refcache doesn't rely on an initial reference holder with a closely correlated life cycle to avoid a degraded state. In some respects this makes it much more general. However, potential multiple passes through the review queue can add substantial overhead to the zero detection process. The latency between initial candidate for free and final release makes it unsuitable for objects with a high rate of turnover. For example, a ten millisecond backlog of network mbufs or VM page structures could incur punitive overhead.

Cache Locality

A simple example of designing for cache locality is packing structures contiguously as opposed to a linked list so that the prefetcher can furnish the next element as a thread iterates through them. At high operation rates, the way in which fields are ordered within a structure can make a measurable performance difference. A 45% increase in brk calls per second was measured when a reorganization of the core memory allocation structure reduced from three to two the number of cache lines for the most commonly accessed fields. Once serialization bottlenecks are eliminated, kernel performance is determined by the frequency of cache misses.

Minimal sharing and cache misses are easily definable ideals for serialization and locality. However, algorithmically defining optimal scheduling for an arbitrary hitherto unseen workload is an unrealizable ideal. Even where the information is present, data structure knowledge and sharing afforded to the scheduler slows down scheduling decisions. When sharing an L2 cache, two communicating threads with a small working set will benefit while two communicating threads with a larger working set will be adversely impacted. A particularly egregious example of where FreeBSD falls down due to thread scheduling on multiple sockets is measured throughput on TCP connections to localhost. On a ~3Ghz single socket system FreeBSD can achieve 50-60Gbps, partly by offloading network processing to *the* `netisr` thread. On a dual socket system of the same clock speed, the measured throughput drops to 18–32Gbps. In the worst case, two communicating processes are on one socket and the `netisr` thread is on the other. Therefore the notification for every packet has to cross the interconnect between sockets. At least on a dual socket, Linux does network processing inline (i.e., no service thread) when doing TCP to localhost. On a single socket Linux would achieve lower throughput than FreeBSD does. However, it achieves a consistent 35Gbps provided both processes are scheduled on the same socket. There are a number of issues to address here:

- the existence of only one `netisr` thread for the entire system
- where the sender, receiver, and `netisr` thread should be scheduled
- how to convey to the scheduler that the three different threads are communicating with each other.

Nonetheless, the key insight here is that latency can determine usable bandwidth and poor scheduling decisions can have a devastating impact on performance when we move from single socket to dual socket.

For users who understand in advance what workloads they will be running, the situation is manageable. The `cpuset` command allows one to assign processor sets to processes, restricting the choices that the scheduler has available to it.

Measuring Scalability

For purposes of this article, scaling measurements will be limited to running the "Will-it-scale" system call microbenchmark suite [Blanchard, 2013] on FreeBSD 11, FreeBSD 12 and Ubuntu 18 (Linux-4.15.1) on two systems—a dual socket EPYC 7601 (2x32 cores @2.2Ghz) and a dual socket Intel Xeon 6130 (2x16 cores @2.1Ghz). The two cannot be directly compared as the EPYC 7601 is a top bin processor retailing for 130% more than the mid-level Xeon 6130. Nonetheless, the more complex EPYC is likely to show very different scaling characteristics and a higher penalty for poor locality.

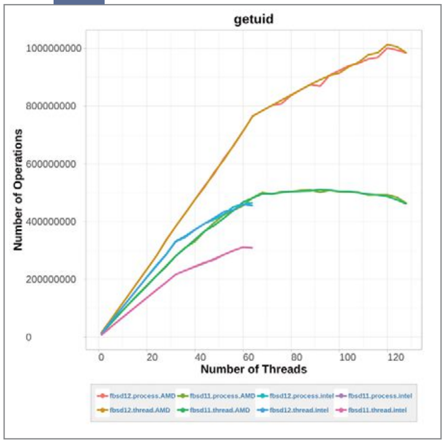
The first thing of note is that the multithreaded variants of most benchmarks scale much more poorly than their multi-process counterparts.The shared address space, file descriptor array, and proc structure all require added locking for the multithreaded case. In the cases where the benchmarks do scale, the curve typicaly flattens at the halfway mark. This is because we move from a regime of one benchmark thread or process per core to oversubscribing the cores and using both hardware threads **(1)**.

There are a number of notable improvements going from FreeBSD 11 to FreeBSD 12.The getuid benchmark shows that system call overhead has been reduced by more than 50%. Page fault performance has improved by 20–80x **(2, 3, 4)**.

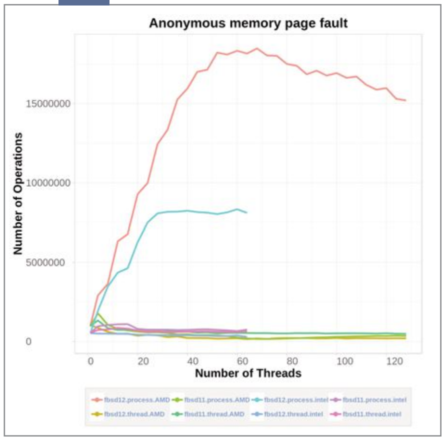
Anonymous memory mmap/munmap of 128MB has improved substantially and currently outperforms Linux **(5, 6)**.

Unix domain socket performance has improved by 19x. Performance previously flattened out at eight hardware threads, but now continues to increase up to 128 hardware threads **(7)**.

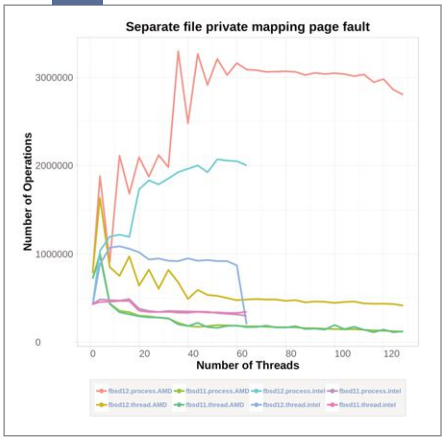
1



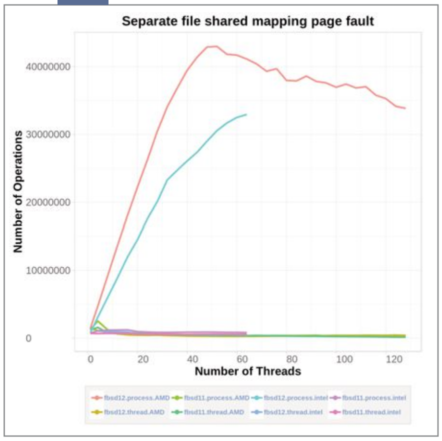
2



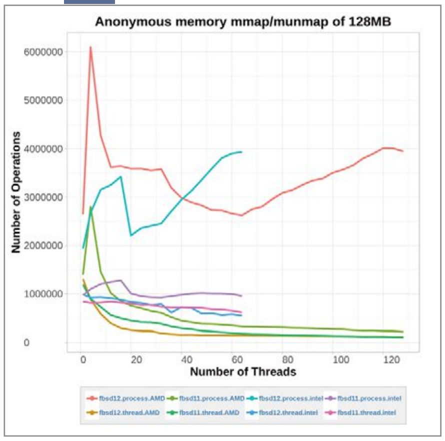
3



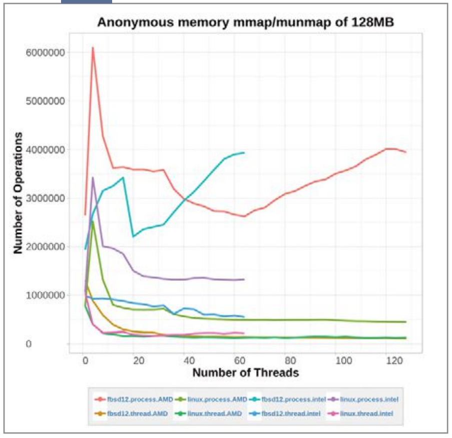
4



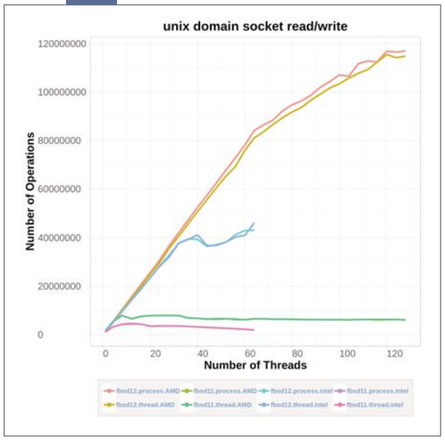
5



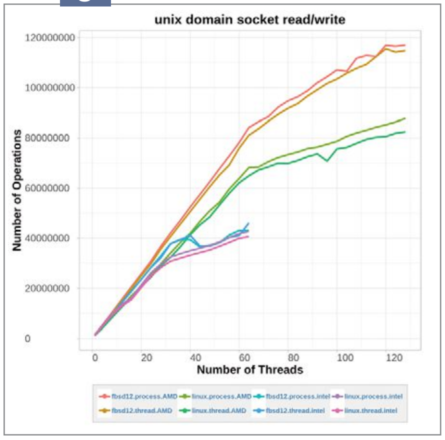
6



7



8



At least as of Linux 4.15 FreeBSD actually scales better than Linux on UNIX sockets **(8)**.

Separate file read still peaks at 32 hardware threads, but it's an 8x improvement **(9, 10)**.

Unfortunately, there are a few areas where underinvestment shows through quite clearly. Although there was an improvement in the brk benchmark from changing handling of swap reservations **(11)**, there are several other system-wide serialization points. Linux scales near linearly here, and at its peak is capable of performing 32x as many brk ops/s **(12)**.

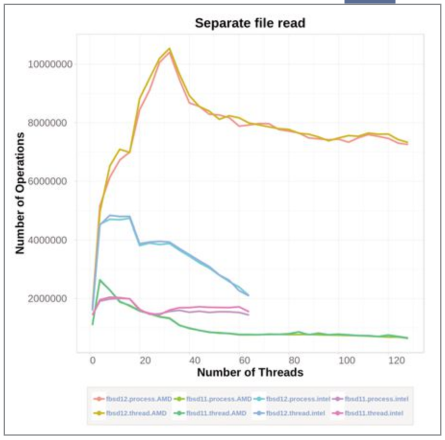
Arguably this isn't that big a deal in practice due to its relative lack of prominence in real world workloads. As a class, the most unsettling difference between FreeBSD and Linux is in filesystem operations. Linux scales near linearly in many cases where FreeBSD stops scaling at four hardware threads **(13, 14, 15, 16)**.

Given more time we would have provided benchmarks with more real world workloads such as the nginx web server serving small static objects, memcached, PostgreSQL, etc.

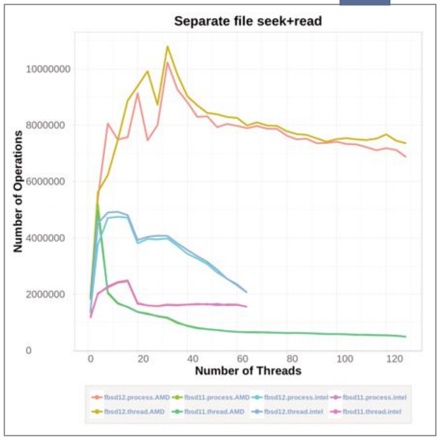
Alternative Approaches

Where do we go from here? Benchmarks can identify how well a system performs but are specific to one workload and configuration. Microbenchmarks are useful for identifying system bottlenecks. However, they don't provide a way to systematically guarantee the absence of scaling limitations. Is there a way to more consistently identify issues? Up until recently the answer was no. However, work in 2014 by Clements [Clements, 2014] built on the notion of disjoint-access-parallel memory systems [Israeli, 94] to rigorously identify limitations in the scalability of both software interfaces and their implementations. This work proposes

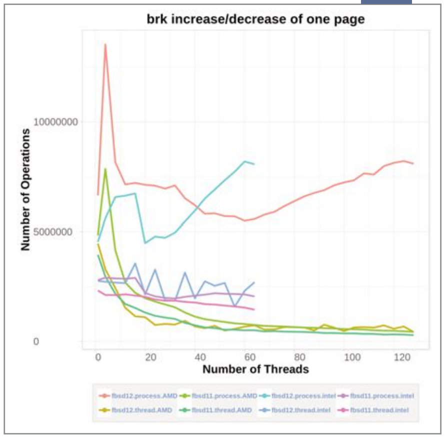
9



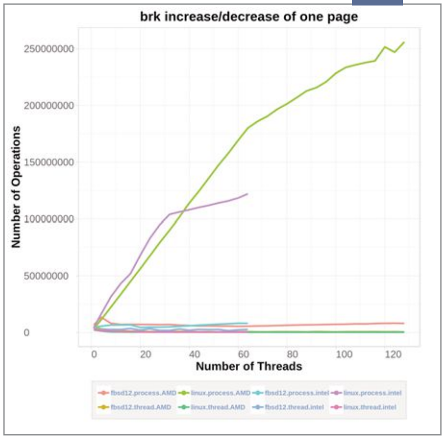
10



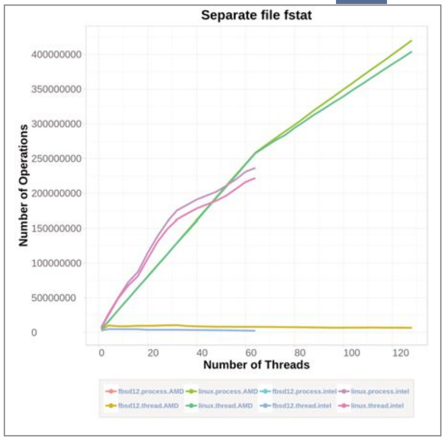
11



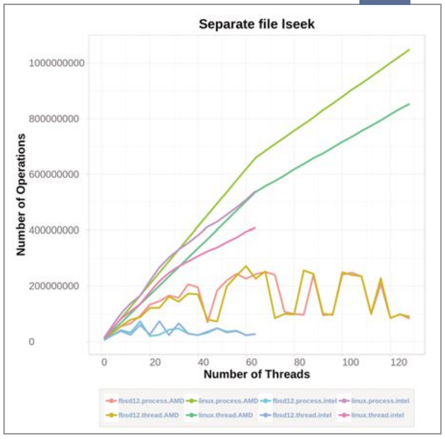
12



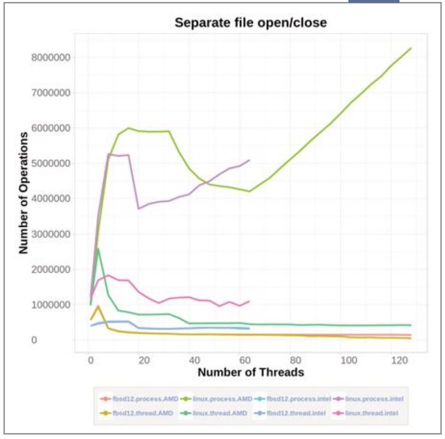
13



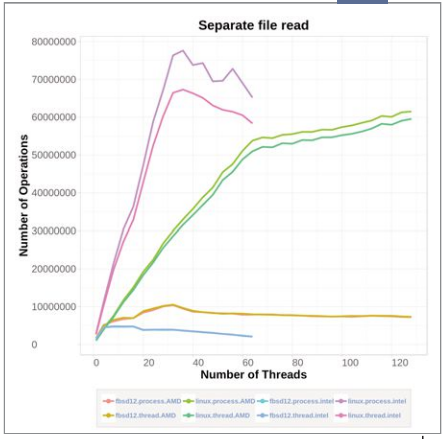
14

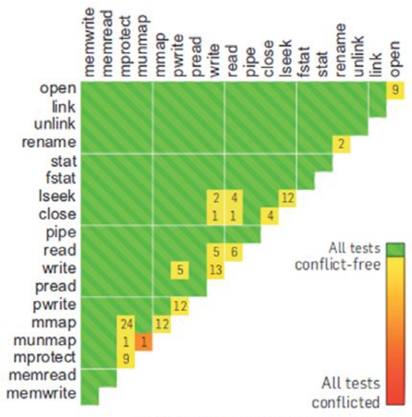


15

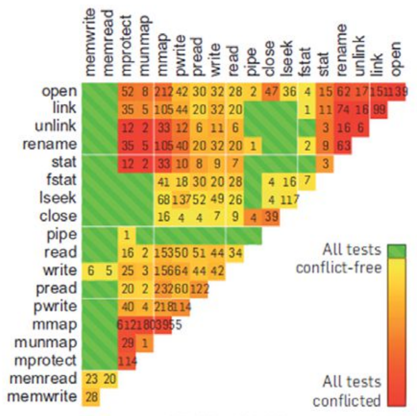


16





sv6 (26,115 of 26,238 cases scale)



Linux (17,206 of 26,238 cases scale)

the *scalable commutativity rule*, which says, in essence, that whenever interface operations commute, they can be implemented in a way that scales. The intuition behind this is simple: when operations commute, their results (both the values returned and any side effects) are independent of order.

He starts by observing that many scalability problems lie not in the implementation, but in the design of the software interface. An interface definition that does not permit two operations to commute enforces serialization between two calls. The POSIX definition of the *open* system call requires that it return the lowest available file descriptor. This means that two calls to open of different files need to be serialized on file descriptor allocation. Some other system calls that have unnecessarily unscalable interfaces are: fork (when immediately followed by exec), stat, sigpending, and munmap.

This is an interesting observation, but the real contribution of the work is developing a tool called *COMMUTER* which:

1. takes a symbolic model of an interface and computes precise conditions for when that interface's operations commute.
2. uses these conditions to generate concrete tests of sets of operations that commute according to the interface model, and thus should have a conflict-free implementation according to the commutativity rule.
3. checks whether a particular implementation is conflict-free for each test case.

He applied this to 18 POSIX system calls to generate 26,238 test cases and used these to compare Linux with sv6, a research OS developed by his group. He found that on Linux 3.8 17,206 cases scale vs 26,115 on sv6. The collection of test cases that failed to scale can be used as a starting point for redesigning subsystems just as the will-it-scale benchmarks have enabled us to identify a much narrower set of issues **(see diagrams left)**.

Porting COMMUTER to work with FreeBSD would be an interesting avenue for future work. ●

BIBLIOGRAPHY

[Attiya, 2011] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., Vechev, M. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: 487-498; http://doi.acm.org/10.1145/1926385.1926442

[Blanchard, 2013] Will-It-Scale benchmark suite. https://github.com/ScaleBSD/will-it-scale.

[Boyd, 2010] S. Boyd-Wickizer, A. T. Clements, . Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI), Vancouver, Canada, Oct. 2010.

[Corbet, 2013] Corbet, J.Per-CPU reference counts, July 2013. https://lwn.net/Articles/557478/

[Clements, 2013] Clements, A. T., M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In Proceedings of the ACM EuroSys Conference, Prague, Czech Republic, April 2013.

[Clements, 2014] Clements, A. T. The scalable commutativity rule: Designing scalable software for multicore processors, Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 2014. [Online]. Available: https://pdos.csail.mit.edu/papers/aclements-phd.pdf

[Culler, 1999] Culler, D. Singh, J. P., Gupta, A. Parallel Computer Architecture - A Hardware / Software Approach, Morgan Kaufman, 1999

[Ellen, 2007] F. Ellen, Y. Lev, V. Luchango, and M. Moir. SNZI: Scalable nonzero indicators. In Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Portland, OR, Aug. 2007.

[Fraser, 2004] Fraser, K. Practical lock-freedom, Ph.D. Thesis, University of Cambridge Computer Laboratory, 2004

[Hackenberg, 2009] D. Hackenberg, D. Molka, and W. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems. MICRO 2009, pages 413–422.

[Hart, 2007] Hart, T. E., McKenney, P. E., Demke Brown, A., Walpole, J. 2007. Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing 67(12): 1270-1285; http://dx.doi.org/10.1016/j.jpdc.2007.04.010

[Herlihy, 2008] Herlihy, M., Shavit, N. 2008. The Art of Multiprocessor Programming. San Francisco: Morgan Kaufmann Publishers Inc.

[Israeli, 1994] Israeli, A., Rappoport, L. Disjoint-access-parallel implementations of strong shared memory primitives. In Proceedings of the 13th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (Los Angeles, CA, August 1994), 151–160.

[McKenney, 2011] McKenney, P. E. 2011. Is parallel programming hard, and, if so, what can you do about it? kernel.org; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

[Michael, 2004] Michael, M. M. Hazard pointers: safe memory reclamation for lock-free objects, IEEE Trans. Parallel Distrib. Syst. 15 (6) (2004) 491–504.

[Molka, 2015] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. 2015. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In Parallel Processing (ICPP), 2015 44th International Conference on.IEEE, 739–748.

[Sorin, 2011] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011.

[Wang, 2016] Q. Wang, T. Stamler, and G. Parmer, "Parallel sections: Scaling system-level data-structures," in Proceedings of the ACM EuroSys Conference, 2016
Appendix A - FreeBSD Serialization Primitives
mutex

APPENDIX A - FreeBSD Serialization Primitives

mutex

Mutex is the most straightforward primitive; ownership is acquired by a thread doing a compare and swap operation on it with a pointer to the acquiring thread's structure. If the lock already contains a thread pointer, it is owned; if not it is free. There are two classes, `MTX_DEF` and `MTX_SPIN`. A `MTX_SPIN` mutex is considered "heavyweight" because it disables interrupts. It can be acquired in any context. While it is held (and with interrupts disabled in general) a thread can only acquire other `MTX_SPIN` locks and cannot allocate memory or sleep. While holding a `MTX_DEF` lock a thread can do anything that would not entail sleeping (acquire either types of mutex, rwlocks, non-sleepable memory allocations, etc.). While waiting to acquire a `MTX_SPIN` mutex a thread will "spin" polling the lock for release by its current holder. While waiting to acquire a `MTX_DEF` a thread will "adaptively spin" on it, polling for release if the current holder is running and being enqueued a `turnstile` if the current holder has been preempted. `Turnstiles` are facility for priority propagation allowing blocked threads to "lend" their scheduler priority to the current lock holder as a mechanism for avoiding priority inversion. In other words, if the blocked thread is higher priority the lockholder's scheduler priority will be elevated to that of the blocked thread.

rwlock

The rwlock extends the semantics of the `MTX_DEF` mutex by supporting two modes – single writer and multiple readers. Its implementation is similar to that of mutex with some additional state assigned to the lower bits of the lock field. In single writer mode it behaves the same as a mutex would. In reader mode, multiple readers can acquire the lock and writers are blocked until all readers drop the lock. In this mode we can no longer efficiently track the lockholders' state so we cannot propagate priority and it is not possible for an acquirer to know if all holders are running. Thus a thread can only spin speculatively.

It supports an arbitrary number of readers, so it's the first primitive developers have traditionally reached for when trying to guarantee existence of fields during a table lookup. However, every read acquisition and release involves an atomic update of the lock. When the lock is shared across core complexes (and thus updates entail cache coherency traffic between LLCs to transition the previous holder's cacheline from modified/exclusive to invalid) its use can quickly become very expensive.

sx

The sx lock is logically equivalent to the rwlock with the critical difference being that a lockholder can sleep. Blocked readers and writers are maintained on sleepqueues and priority propagation is not done.

rmlock

Like the rwlock and sx, the rmlock "read mostly lock" is a reader/writer lock. Its critical difference is that acquisition for read is extremely fast and does not involve any atomics. Acquisition for write is extremely expensive. In its current incarnation it involves a system wide IPI to all other cpus. This is actually a reasonable primitive for guaranteeing existence if updates are infrequent enough. It is easy to reason about, having the same semantics as familiar rwlocks.

lockmgr

Lockmgr is something of an anachronism. It has some unique features required by the VFS (virtual file system) layer. It is generally not a bottleneck in today's code and its idiosyncracies are outside the scope of what I hope to touch on.

epoch

The epoch primitive allows the kernel to guarantee that structures protected by it will remain live while a thread is in an epoch section. Executing `do_stuff()` in an epoch section looks something like:

```
epoch_enter(global_epoch);
do_stuff(); ...
epoch_exit(global_epoch);
```

A thread deleting an object referenced within an epoch section can either synchronously wait for all threads in an epoch section during the current epoch plus a grace period by calling `epoch_wait(epoch)`, or it can enqueue the object to be freed at a later time using `epoch_call(epoch, context, callback)`, allowing a service thread to confirm – at lower cost than a synchronous operation – that a grace period has elapsed. In many respects the read side of epoch has similar characteristics to the read side of an `rmlock`. However, it does not provide a mutual exclusion guarantee. Modifications to an epoch protected data structure can proceed in parallel with readers. Modifications do typically need to be explicitly serialized with respect to each other. Thus a mutex is used to protect a writer against other writers. Although its implementation and the performance trade-offs are completely different from Linux's RCU, it largely supports the same programming design patterns. There are two variants of epoch, preemptible and non-preemptible. A non-preemptible epoch is lighter weight but does not permit the calling thread to acquire any lock type other than `MTX_SPIN` mutexes. Epoch is new in FreeBSD 12. It is essentially in-kernel scaffolding built around ConcurrencyKit's epoch (Epoch Based Reclamation) API.

Matt Macy is a Consulting Kernel Engineer and FreeBSD committer. Recently he has focused on networking performance, kernel scalability, and ZFS.

APPENDIX B Will-It-Scale Results

The complete results for Will-It-Scale for FreeBSD 11 vs FreeBSD 12 and FreeBSD 12 vs Ubuntu 18 (Linux 4.15) can be found at: https://github.com/ScaleBSD/scalebsd.github.io/tree/master/media/freebsd_processor_scaling

SEE TEXT ONLY

new faces of FreeBSD

BY DRU LAVIGNE

This column aims to shine a spotlight on contributors who recently received their commit bit and to introduce them to the FreeBSD community. In this month's column, the spotlight is on Breno Leitao who received his src bit in May.

Tell us a bit about yourself, your background, and your interests.

● Breno: I am passionate about open source since I started using it in 1997. Until that time, I was a MS-DOS/Windows user who had very little control of my own machine. After the discovery of open source, I started to see computers from a different viewpoint, which blasted my mind and I was finally able to understand how a computer works, and I felt, for the very first time, that I really had control of my own machine. At that point, I didn't have the skill to change many things at depth, but I had the most important thing, the source code to do so.

I started using different distros of Linux and BBSes and I was a happy guy. Around 1999, I owned a 486 DX 66MHz, the Internet was still new, and downloading music from the Internet was one of the coolest things you would use the Internet for. When I was finally able to download my first MP3 file from the Internet using the PPP protocol, I discovered that my system was not fast enough to play an MP3 file without breaking up.

I didn't feel depressed with this discovery and decided to try to do anything to be able to play an MP3 file on the only computer I had access to. Reading things on the Internet, I heard about a super fast OS named FreeBSD.

How did you first learn about FreeBSD, and what about FreeBSD interested you?

● Breno: In order to be able to play an MP3 file in my computer, I decided to wipe out my system and install FreeBSD. It was version 4 at that time, and I thought it would be a good idea to test something new. After the installation, I started the MP3 decoder, and no luck, the music was still breaking up on this new OS.

At that time, I decided to dig further and look into the source code. Of course I was not able to understand much at that time, but I was able to understand something very important that I was not aware of until that point. The kernel was built using major features that you can disable if you do not use them.

I could reconfigure my kernel, avoiding code that I was not using, aiming to speed up the MP3 player.

After playing with it for a while, doing some recompilation with different compiler flags, removing multi-user capability, I was finally able to play the MP3 I had downloaded. That was a great moment in my life!

From that day on, I started to play with other BSDs, and I moved to NetBSD for a while since it was the only OS I found that ships all the packages in the archive on 6 CDs. So, I downloaded and burned the ISOs at the university to have the full archive at home without the long download time.

At that same time, I found the first edition of *The Design and Implementation of the FreeBSD Operating System* at the university library, and I was amazed. I rented it and started to read it, but I didn't have the OS concepts to fully understand it. Got it again and again until I bought it and, 15 years later, I am still reading it.

How did you end up becoming a committer?

● Breno: After I left the university I was quite fortunate to find a job at IBM to do OS development, mostly Linux on Powerpc. After several years using mostly Linux, I decided to try FreeBSD on the Power systems. That brought me the joy I used to have when I was a kid, and after some hacking I got FreeBSD running on the pSeries platform, and the BSD flame was rekindled.

After a while, I talked to the FreeBSD/powerpc maintainers and they were so cool and patient with my silly questions/code that I finally found myself at home. Feeling at home was quite important for me to be able to start sending patches. In one hacking night I was invited by Justin Hibbits and Nathan Whitehorn to become a committer. Even before that day, they had been very patient with me and I would like to take this opportunity to say "Thanks."

How has your experience been since joining the FreeBSD Project? Do you have any advice for readers who may be interested in also becoming a FreeBSD committer?

• Breno: The experience is unique, mainly because the developers are an IRC message away, which is fascinating to me. I can ping the guy who wrote any specific code and ask (mostly) silly questions, and they've been very cooperative, and so far, very helpful. Having senior developers be accessible and supportive is a rare experience for me, thus, the uniqueness of the project.

I am not very good at giving advice, but the fact that I've pestered the Powerpc maintainers with questions (all day sometimes) was key for me to be able to solve important problems, and be asked to become a committer. •

DRU LAVIGNE is the Director of Storage Engineering at iXsystems, author of *BSD Hacks* and *The Best of FreeBSD Basics*.

SEE TEXT ONLY

WeGetletters by Michael W Lucas

Hey, FJ Letters Dude,
Which filesystem should I use?
—FreeBSD Newbie

Dear FreeBSD Newbie,
First off, welcome to FreeBSD. The wider community is glad to help you.

Second, please let me know who told you to start off by writing me. I need to properly… "thank" them.

Filesystems? Sure, let's talk filesystems.

Discussing which filesystem is the worst is like debating the merits of two-handed swords as compared to lumberjack-grade chainsaws and industrial tulip presses. While every one of them has perfectly legitimate uses, in the hands of the novice they're far more likely to maim everyone involved. It doesn't matter what operating system you use: FreeBSD, any BSD, Linux, Windows, illumos, whatever. Filesystems are the literal worst.

I mean, let's look at memory filesystems. The base idea, taking a chunk of memory and using it for temporary storage, seems sound enough. Most non-virtual computers these days have more than enough memory that they can blow a few gigabytes for a speedy /tmp or perhaps even compiler scratch space. Configuring poudriere to use memory for temporary files will vastly accelerate your package builds.

But FreeBSD has two different memory filesystems, mfs(5) and tmpfs(5). Old-fashioned MFS blats a UFS filesystem down on top of a chunk of memory. It's fast, sure. But any space MFS uses is unavailable for other use as long as the filesystem exists. Suppose you create a 5 GB /tmp with MFS, copy 4.9 GB to it, and erase it. That 4.9 GB is still tied up. You can instruct MFS to free unused memory by enabling TRIM with tunefs(8), but nobody bothers with that.

The newer alternative, tmpfs, is specifically designed for temporary filesystems. A default tmpfs has a maximum size equal to the system memory plus the system's swap space. "How much memory do you have? Give it to me." Be sure to specify the size= flag when you create a tmpfs, or be careful to monitor tmpfs space use. Not that you'll configure your monitoring system to watch tmpfs, because it's temporary.

And no matter what, one day you'll forget that you used memory space as a filesystem. You'll stash something vital in that temporary space, then reboot. And get really annoyed when that vital data vanishes into the ether.

Some other filesystems aren't actively terrible. The device filesystem devfs(5) provides device nodes. Filesystems that can't store user data are the best filesystems. But then some clever sysadmin decides to hack on /etc/devfs.rules to change the standard device nodes for their special application, or /etc/devd.conf to create or reconfigure device nodes, and the whole system goes down the tubes.

Speaking of clever sysadmins, now and then people decide that they want to optimize disk space or cut down how many copies of a file they need to maintain by reusing a partition or dataset elsewhere on the system. FreeBSD's nullfs(5) lets you mount a partition multiple times, essentially recycling the same chunk of disk space. Folks who use a bunch of jails use read-only nullfs mounts to have a single FreeBSD base install support multiple jails.

FreeBSD's unionfs(5) lets you merge multiple filesystems. Many people successfully use unionfs to provide custom views of a filesystem, again for jails. Unionfs is perhaps the least popular filesystem in the FreeBSD ecosystem though. I know several developers who won't go near it. I know others who say it's perfectly safe. All I know is, backups are good.

Network filesystems? Oh please. A dedicated 6GB/s SATA controller is always going to outperform anything that runs over gigabit Ethernet, especially if you're using that same network interface to manage the host on. Yes, six gig is more than one gig—but that comparison also has bits versus bytes. You're looking at a 48-fold difference in optimal throughput. And always remember that not all network switches are created equal. I have a whole stack of so-called "gigabit" switches that utterly refuse to pass more than a quarter gigabit a second.

I must unwillingly concede that FreeBSD's new iSCSI stack is rock solid. And FreeBSD's NFS implementation is among the best in the world. Many people use these in high-performance applications… but they're still networked filesystems. These people battering them in production have top-notch network cards and switches that live up to the hype. If you ask on the mailing lists or forums, they'll offer their advice.

FreeBSD has excellent support for the new NFSv4 protocol. While earlier versions of NFS interoperate pretty well and have identical behavior, NFSv4 is a whole different beast with different semantics. You really need to do some reading before deploying it. NFSv4 does have an extensive access control list system that lets you perfectly implement the worst abominations a large corporation's IT department can dream up, so that's something.

You'll occasionally see mentions of the process filesystem, procfs(5). Many FreeBSD developers really, really don't want procfs to exist. When I documented a need for procfs in the 2018 version of ***Absolute FreeBSD***, technical reviewer John Baldwin rewrote ps(1) to make procfs unnecessary. As far as I can tell, the quickest way to goad a FreeBSD developer into action is to need /proc.

Autofs(5) was written for desktop users. It automatically identifies filesystems and mounts them for you. If you enable autofs and plug in a USB drive, the various partitions and labels on the drive will appear as directories in /media. Going into one of those directories will automatically mount that partition. Similarly, autofs makes NFS mount points available in /net. Listing the contents of /net/fileserver displays all the NFS mount points on the host fileserver, and going into one of those directories automatically mounts the share. It's still using a networked filesystem though, so it'll almost certainly end in tears.

In the defense of all of FreeBSD's filesystems, though, I must say: at least they're not EXTFS. Although FreeBSD supports extfs(5) as well, so that's not much help.

Really, the only smart move with filesystems is not to play.

FJ Letters Dude,
I meant, I'm looking at the installer and it wants to know if I want to use UFS or ZFS? And George Neville-Neil said you needed letters.

—FreeBSD Newbie

Oh!
Use ZFS, unless you can't.

As a new user, don't use ZFS on systems with less than two GB of RAM. Four GB or more would be wiser. Don't use ZFS on non-64-bit platforms.

Some virtualization systems don't properly label disk images during migration from one host to another. ZFS pools migrated on such systems won't boot. If you're running on a virtualization platform, test migration on a ZFS host before deploying it everywhere.

And thanks for the tip. Next time I run into GNN, we'll discuss his unfortunate tendency to encourage people.

Michael W Lucas (https://mwl.io) is the author of too many books, including *Absolute FreeBSD*, *FreeBSD Mastery: Specialty Filesystems*, and *git commit murder*.

Send your questions to letters@freebsdjournal.com. Letters will be answered in the order in which they amuse, annoy, or inspire the columnist, and may be edited for his own purposes.

SEE TEXT ONLY

svn UPDATE

by Steven Kreuzer

We have entered the 12.0 release cycle! The code freeze has begun and the 12.0-RELEASE announcement is scheduled for November 13. It's likely that by the time you read this column, the releng/12.0 branch will have been created.

One thing I am thrilled to see is that more of the userland tools are gaining libxo support. What I have always loved about the Unix command line is the ability to pipe output from one command to another to perform some manipulation before sending it off to another command for some final processing. My biggest pet peeve is that all this output doesn't follow any real "standard" and most of the time you find yourself having to parse blocks of text, looking for a certain string, counting a few lines down from there and performing other acts of pure insanity. libxo solves this by making it easy to generate text, XML, JSON, and HTML output using a common set of function calls. It is a welcome addition to the tree.

pf: Support "return" statements in passing rules when they fail—

https://svnweb.freebsd.org/changeset/base/335569

Normally pf rules are expected to do one of two things: pass the traffic or block it. Blocking can be silent - "drop", or loud - "return", "return-rst", "return-icmp". Yet there is a third category of traffic passing through pf: packets matching a "pass" rule but, when applying the rule, fail. This happens when a redirection table is empty or when src node or state creation fails. Such rules always fail silently without notifying the sender. Allow users to configure this behavior too so that pf returns an error packet in these cases.

Introduce arm64 linuxulator stubs—

https://svnweb.freebsd.org/changeset/base/335333

This provides stub implementations of arm64 Linux vdso and machdep, ptrace, and futex sufficient for executing an arm64 Linux 'hello world' binary.

Make UMA and malloc(9) return non-executable memory in most cases—

https://svnweb.freebsd.org/changeset/base/335068

Most kernel memory that is allocated after boot does not need to be executable. There are a few exceptions. For example, kernel modules do need executable memory, but they don't use UMA or malloc(9). The BPF JIT compiler also needs executable memory and did use malloc(9) until r317072. This change makes malloc(9) return non-executable memory unless the new M_EXEC flag is specified. After this change, the UMA zone(9) allocator will always return non-executable memory and a KASSERT will catch attempts to use the M_EXEC flag to allocate executable memory using uma_zalloc() or its variants.

Flip the default interpreter to Lua—

https://svnweb.freebsd.org/changeset/base/338050

After years in the making, lualoader is ready to make its debut. Both flavors of loader are still built by default, and may be installed as /boot/loader or /boot/loader.efi as appropriate either, by manually creating hard links or using LOADER_DEFAULT_INTERP as documented in build(7).

Add libxo(3) support to lastlogin(8)—

https://svnweb.freebsd.org/changeset/base/338353

Add libxo(3) support to last(1)—

https://svnweb.freebsd.org/changeset/base/338352

Speed up vt(4) by keeping a record of the most recently drawn character and the foreground and background colors—

https://svnweb.freebsd.org/changeset/base/338316

In bitblt_text functions, compare values to this cache and don't redraw the characters if they haven't changed. When invalidating the display, clear this cache in order to force characters to be redrawn; also force full redraws between suspend/resume pairs since odd artifacts can otherwise result. When scrolling the display (which is where most time is spent within the vt driver) this yields a significant performance improvement if most lines are less than the width of the terminal, since this avoids redrawing blanks on top of blanks. On a c5.4xlarge EC2 instance (with emulated text mode VGA), this cuts the time spent in vt(4) during the kernel boot from 1200 ms to 700 ms; on my laptop (with a 3200x1800 display) the corresponding time is reduced from 970 ms down to 155 ms.

Make it possible for init to execute any executable, not just sh(1) scripts—
https://svnweb.freebsd.org/changeset/base/337321

This means one should be able to eg rewrite their /etc/rcin Python.

Add the ability to run bhyve(8) within a jail(8)—
https://svnweb.freebsd.org/changeset/base/337023

This patch adds a new sysctl(8) knob "security.jail.vmm_allowed"; by default this option is disable.

Extend loader(8) geli support to all architectures and all disk-like devices—
https://svnweb.freebsd.org/changeset/base/336252

This moves the bulk of the geli support from lib386/biosdisk.c into a new geli/gelidev.c which implements a devsw-type device whose dv_strategy() function handles geli decryption. Support for all arches comes from moving the taste-and-attach code to the devopen() function in libsa. After opening any DEVT_DISK device, devopen() calls the new function geli_probe_and_attach(), which will "attach" the geli code to the open_file struct by creating a geli_devdesc instance to replace the disk_devdesc instance in the open_file. That routes all IO for the device through the geli code. A new public geli_add_key() function is added to allow arch/vendor-specific code to add keys obtained from custom hardware or other sources. With these changes, geli support will be compiled into all variations of loader(8) on all arches because the default is WITH_LOADER_GELI.

Add bhyve NVMe device emulation—
https://svnweb.freebsd.org/changeset/base/335974

The initial work on bhyve NVMe device emulation was done by the GSoC student Shunsuke Mie and was heavily modified in performance, functionality and guest support by Leon Dang.

Enable options NUMA in GENERIC and MINIMAL—
https://svnweb.freebsd.org/changeset/base/338602

Switch the default pager for most commands to less—
https://svnweb.freebsd.org/changeset/base/337497

Sometimes less is more, and now it is consistent across base.

STEVEN KREUZER is a FreeBSD Developer and Unix Systems Administrator with an interest in retrocomputing and air-cooled Volkswagens. He lives in Queens, New York, with his wife, daughter, and dog.

SEE TEXT ONLY

conference REPORT

BSDCan and Dev Summit by Christian Schwarz

This year I attended my first BSDCan conference, after having enjoyed the last two EuroBSDCons in Belgrade and Paris. Luckily, my talk proposal on zrepl was accepted and thus flight and accomoda-tion costs were covered.

As Benedict Reuschling (bcr@) and I only live ~100km apart, we coordinated travel, which commenced on June 1 with a direct flight from FRA to YOW. At FRA, we also met Kirill Ponomarev (krion@) and enjoyed the amenities of the Air Canada lounge. After pain-free border control and a short taxi ride, we made ourselves comfortable at U90 and attempted to stay awake to fight the jet lag.

The following day included sightseeing in Ottawa, great food at Byward Market, and a bus ride around the city. Incidentally, it wasn't until we were already well into the tour that we realized sun screen or a hat could have been put to good use under the almost cloud-free sky.

Benedict and I had coordinated with Dru Lavigne and Warren Block to visit Montreal via the train from Ottawa. With beautiful weather on the first day, we walked around town to the old port and later took a cab up Mount Royal where we enjoyed a nice view of the city. The following day was a demonstration of how quickly Montreal weather can change. Under heavy rain, we had breakfast in a cafe and stayed inside to hack on our projects while waiting for the rain to stop. Sadly, the weather would not change, and so we finally wandered through the enormous collection of underground shopping malls in search of poutine, which I had yet to experience. My verdict: heavy stuff, but I can see its appeal during a cold Canadian winter.

Back in Ottawa, I used the Wednesday before the Dev Summit to finish my slides and rehearse the talk. In the evening, people started flooding the Royal Oak, where I got to discuss (and resolve!) a sysrc(8) issue with Devin Teske.

Thursday and Friday was Dev Summit: In contrast to what I had experienced in Paris the previous autumn, this one involved a little less hacking and more project coordination, such as the updates from core@ and secteam@ as well as the discussion of the FreeBSD 12 roadmap. I also attended the OpenZFS working group—the topic I am currently most interested in. We had many great discussions and I suggested some of my ideas to improve usability of ZFS with other automated tools. Although the evenings of both days were spent at the hacker lounge, there was arguably more discussion, planning, and socializing than getting actual commits done. However, I can write code the other 51 weeks of the year, so that was just fine!

The actual conference began on Saturday. Although I missed Benno Rice's keynote, I attended Sara Hartse's excellent talk on fast clone deletion in ZFS, followed by Rod Grimes's talk (or rather discussion) on possible performance improvements for ZFS send and receive. After lunch, I listened to Brooks Davis's talk on a possible replacement for mmap and Sean Chittenden's presentation on hosting virtual private clouds powered by bhyve / FreeBSD, which offered some insight into the kernel drivers and interfaces they had to modify to enable more than 30Gbps tunneled network throughput and better integration into their control plane.

Day 2 started with Matt Ahrens's talk on ZFS device removal and raidz expansion—both highly anticipated features in the community. Subsequently, I attended Stefan Grönke's talk on libiocage, the Python library that started as a rewrite of iocage internals, but now provides a pythonic, high-level abstraction over jails. After lunch, I started getting nervous about my talk, scheduled for later in the afternoon, but once I got over the initial mental hurdle, the talking, timing, and live demo went smoothly, and the subsequent comments from the audience gave me valuable feedback for the future development of zrepl. I stayed for Kirk McKusick's presentation on the evolution of FreeBSD governance, which was both an entertaining and informative overview of the project's organizational and social history and current structure. The closing session, including the renowned charity auction, put an end to the official part of the confierence and was followed by a mass migration to the Red Lion, where the closing social event took place.

Needless to say, all the hallway-track time and the conversations at the various social events were just as valuable to me as the actual conference. I met many very nice people, learned a lot, and extended my to-do wish-list for FreeBSD by factor 2x. I will certainly try to come back next year!

Christian Schwarz is a computer science student, currently pursuing his master's degree in Karlsruhe, Germany. His main interests are operating systems and compiler technology, although he is drawn to all kinds of software systems. He is the developer of zrepl (https://zrepl.github.io/), a new- general-purpose solution for ZFS replication.

SEPT/OCT

BY DRU LAVIGNE

2018 Events Calendar

The following BSD-related conferences will take place during the last quarter of 2018.

MeetBSD • Oct 19 & 20 • Santa Clara, CA

https://www.meetbsd.com/ • MeetBSD California is a biennial BSD Unix conference that takes place in Silicon Valley. This event is a mixed unConference format event featuring both scheduled talks and community-driven events such as birds-of-a-feather meetings and speed geeking sessions.

All Things Open • Oct 21–23 • Raleigh, NC

http://allthingsopen.org/ • All Things Open is the largest open source/open tech/open web conference on the East Coast, and one of the largest in the United States. It regularly hosts some of the most well-known experts in the world as well as nearly every major technology company. More than 3,200 from 34 states and 19 countries participated in 2017. This year between 3,500 and 4,000+ are expected. The Foundation is pleased to be a media partner.

LISA • Oct 29–31 • Nashville, TN

https://www.usenix.org/conference/lisa18 • LISA is the premier conference for operations professionals, where sysadmins, systems engineers, IT operations professionals, SRE practitioners, developers, IT managers, and academic researchers share real-world knowledge about designing, building, securing, and maintaining the critical systems of our interconnected world. There will be a FreeBSD booth in the Expo area.